SBC6020 Linux Application Development Guide

Rev. 2.0

Release: 2010-5-11

Revision history

Rev	Date	Description
1.0	20100115	Initial version
2.0	20100508	3.7 CAN Bus Test 4.3.3 The Second Booting Stage-U-boot
2.0	20100511	Delete about the dataflash

Table of Contents

SBC6020 Linux Application Development Guide 1
Chapter 1 Overview of Linux system 4
Chapter 2 Getting Start with SBC6020 5
Chapter 3 Test Tutorials of the SBC6020 6
3.1 RTC Test 6
3.2 USB Host Test 6
3.3 CF card test 7
3.4 RS-485 test 7
3.5 GPIO Test 8
3.6 UART Test 8
3.7 CAN Bus Test 8
3.8 ADC Test 9
3.9 Beep Test 9
3.10 Net Test 9
Chapter 4 Setting the Development Environment 10
4.1 Install the Cross Compiler 10
4.2 Setting the Cross-compiler Environment 10
4.3 Compiler the System of the SBC6020 10
4.4 System Customization 12
Chapter 5 Update the System Image 15
5.1 System Image Map 15
5.2 Setting the Environment of Burning Image 15
5.3 Burning System Image 16
Chapter 6 Application Software Development 32
6.1 CAN BUS Application Development 32
6.2 UART Application Development 37
6.3 Watchdog Application Development 39
6.4 GPIO Application Development 40
6.5 Beep Application Development 41
6.6 ADC Application Development 42

Chapter 1 Overview of Linux system

SBC6020 BSP mainly used for customization Linux for SBC6020 hardware platform, you can do secondary development based on the development kit. The SBC6020 CD provides the BSP packet that refers to table 1-1 content.

Table 1-1 specification of SBC6020 BSP

Name		Remark
BIOS	Bootstrap	NANDFLASH
	u-boot	NANDFLASH
		SAM-BA and NET
Kernel	Linux-2.6.30	ROM/CRAM/EXT2/EXT3/FAT/NFS/ JFFS2/YAFFS2..etc
Device Driver	Serial	CPU DEBUG and five ways Serial
	RS-485	One way RS-485
	NET	Two-way 10/100M Ethernet driver
	Flash	Nand Flash driver
	USB Host	USB Host driver
	USB Device	USB Device driver
	Beep	Beep driver
	LED	User LED driver
	GPIO	GPIO driver
	Watchdog	Watchdog driver
	CAN	CAN BUS driver
	CF Card	True IDE mode
	EEPROM	256 bytes EEprom
	ADC	4 channels 10bit ADC
Server	Web Server	Apache

Chapter 2 Getting Start with SBC6020

Note:If you use the HyperTerminal on the PC when SBC6020 is booting, please configure your connection software like this:
Baud rate:115200
Data: 8bit;
Parity:None
Stop: 1bit
Flow control: None

Getting Start from Nandflash

The development board support booting from nandflash The development board have burnedthe system image of the nandflash. If you want update the nandflash image,you can refer to the "*Chapter 5 Update the System Image*".

After update the image,close the J8 and J11 by the jumper cap. Power on, the HyperTerminalon will output system starting information from the nandflash.

Note: The system image update can refer to【Chapter 5 Update the System Image】

Chapter 3 Test Tutorials of the SBC6020

3.1 RTC Test

The development board has hardware clock chip used for store/restore system time. You can opera the RTC like this:

1. View system clock information

```
~ $ date
Fri Jul 10 09:25:13 UTC 2009
```

2. Change the date, like July 10 09:01 2009.

```
~ $ date 071009012009
Fri Jul 10 09:01:00 UTC 2009
```

3.Save the system clock to RTC

```
~ $ hwclock -w
```

4. View RTC clock information

```
~ $ hwclock
Fri Jul 10 09:01:11 2009    0.000000 seconds
```

5. Restore the RTC backup to system.

```
~ $ hwclock -s
~ $ date
Fri Jul 10 09:03:17 UTC 2009
```

3.2 USB Host Test

There are two USB host port on the development board. Test the USB host will need a USB device (For example, an usb disk). When you plug in USB device, system will print related Information on your HyperTerminal.

```
There'll display like this if you plug in USB device:usb 1-1: USB disconnect, address 2
usb 1-1: new full speed USB device using at91_ohci and address 3
usb 1-1: configuration #1 chosen from 1 choice
scsi2 : SCSI emulation for USB Mass Storage devices
scsi 2:0:0:0: Direct-Access         Generic   USB   SD Reader     0.00 PQ: 0 ANSI: 2
sd 2:0:0:0: [sdb] 7744512 512-byte hardware sectors (3965 MB)
sd 2:0:0:0: [sdb] Write Protect is off
sd 2:0:0:0: [sdb] Assuming drive cache: write through
sd 2:0:0:0: [sdb] 7744512 512-byte hardware sectors (3965 MB)
sd 2:0:0:0: [sdb] Write Protect is off
sd 2:0:0:0: [sdb] Assuming drive cache: write through
 sdb: sdb1
sd 2:0:0:0: [sdb] Attached SCSI removable disk
sd 2:0:0:0: Attached scsi generic sg1 type 0
```

Prompted by the above information can be seen that usb disk device was identified as sdb1.

1.Users can opera the usb disk after mount u-disk by following command, because Linux system is request must have a directory was used by usb disk.For example, you can mount it into /mnt directory and set vfat format.

```
~ $    mount -t vfat /dev/sdb1 /mnt/
```

2.Enter into the directory and view files.

```
~ $    cd /mnt/
/mnt $    ls
```

3.Umount the usb disk

```
/home/app/can $    cd /
~ $    umount /mnt/
```

Note: usb disk will identify as sdb device by default, you can seen it from system prompt.So as SD card test.

3.3 CF card test

The CF card using the True IDE interface does not support the hot-swap mode, CF card must be inserted before power on. After power-printed from a serial terminal system message as follows:

```
ata1.00: CFA: SanDisk SDCFH2-2048, HDX 4.04, max MWDMA2
ata1.00: 4001760 sectors, multi 0: LBA
ata1.00: configured for PIO4
scsi 0:0:0:0: Direct-Access     ATA      SanDisk SDCFH2-2 HDX    PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 4001760 512-byte hardware sectors: (2.04 GB/1.90 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
 sda:
sd 0:0:0:0: [sda] Attached SCSI disk
```

From the above terminal output information can be seen, CF card was identified as sda. You can use the following command to its operation:

1.Users can opera the CF card after mount CF card by following command, because Linux system is request must have a directory was used by CF card.For example, you can mount it into /mnt directory and set vfat format.

```
~ $    mount -t vfat /dev/sda /mnt/
```

2.Enter into the directory and view files.

```
~ $    cd /mnt/
/mnt $    ls
```

3.Umount the CF card

```
~ $    umount /mnt/
```

3.4 RS-485 test

The RS-485 take up serial device /dev/ttyS2. Test methods, such as serial port can be tested solution, but only in the hardware connection is different.

3.5 GPIO Test

There are 8 GPIO ports on the development board, index 0 to 7. Users can enter following command to opera each GPIO, All about GPIO hardware connects can refer to "Hardware manual of SBC6020".

1. Set the first GPIO output.

```
/home/app $    ./gpioctl -o 0
```

2. Set the first GPIO as high level.

```
/home/app $    ./gpioctl -s 0
```

3. Set the first GPIO as low level.

```
/home/app $    ./gpioctl -c 0
```

4. Set the first GPIO input.

```
/home/app $    ./gpioctl -i 0
```

5. Get the first GPIO's input value.

```
/home/app $    ./gpioctl -g 0
```

3.6 UART Test

There are seven serial ports on the development board, which CPU has 4, and each device named for Linux is "ttyS0, ttyS1, ttyS2, ttyS3, ttyS4, ttyS5 and ttyS6", "ttyS0" is use for debug. "ttyS2" is used for the RS-485. You can do test with following commands. All about Serial port hardware connects can refer to "Hardware manual of SBC6020".

```
/home/app $ ./com –d /dev/ttyS1 -s 1234567890
```

Note: pressing the Ctrl+C then exit the test program.

3.7 CAN Bus Test

There is only one CAN device on the development board, which need connect other CAN BUS device (for example: CANET100, a CAN test device). All about CAN BUS hardware connects can refer to "Hardware manual of SBC6020".

1. CAN receive test:

```
~ $    cd /home/app/can/
/home/app/can $    ./baud /dev/can0 100
/home/app/can $    ./receive can0
```

This is received data:

```
Received with ret=1:     1246986251.249889 id=1/0x00000001
            len=8 flags=0x00 : sD : msg= 00 11 22 33 44 55 66 77
Received with ret=1:     1246986252.701898 id=1/0x00000001
            len=8 flags=0x00 : sD : msg= 00 11 22 33 44 55 66 77
Received with ret=1:     1246986254.069926 id=1/0x00000001
            len=8 flags=0x00 : sD : msg= 00 11 22 33 44 55 66 77
```

2. CAN transmit test.

```
/home/app/can $ ./transmit can0
```

Note: pressing the Ctrl+C then exit the test program.

3.8 ADC Test

Development board is expansion of an 4-channel 10-bit ADC. All about ADC hardware connects can refer to "Hardware manual of SBC6020".

```
/home/app$ ./adc /dev/at91adc0
```

It will print the 4-cannel ADC sample value on the terminal.

3.9 Beep Test

There have a Beep driving circuit on the development board, all about Beep hardware connects can refer to "Hardware manual of SBC6020".

Execute the following command; it will ring for every 1 second.

```
/home/app$ ./beep –r 1 –q 1
```

3.10 Net Test

There have a DM9000 10/100M Ethernet on the development board, users can use it connect to network, and use following command to test:

```
~ $   ifconfig eth0 192.192.192.200
~ $   ping 192.192.192.105
PING 192.192.192.105 (192.192.192.105): 56 data bytes
64 bytes from 192.192.192.105: icmp_seq=0 ttl=64 time=0.5 ms
64 bytes from 192.192.192.105: icmp_seq=1 ttl=64 time=0.3 ms
64 bytes from 192.192.192.105: icmp_seq=2 ttl=64 time=0.3 ms
64 bytes from 192.192.192.105: icmp_seq=3 ttl=64 time=0.3 ms
64 bytes from 192.192.192.105: icmp_seq=4 ttl=64 time=0.3 ms
64 bytes from 192.192.192.105: icmp_seq=5 ttl=64 time=0.3 ms
64 bytes from 192.192.192.105: icmp_seq=6 ttl=64 time=0.3 ms

--- 192.192.192.105 ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max = 0.3/0.3/0.5 ms
~ $
```

Chapter 4 Setting the Development Environment

Before development the SBC6020,users must setup arm-linux cross development environment. Following the example of Ubuntu operating system(Linux).

4.1 Install the Cross Compiler

If you have not installed arm-linux-gcc, copy **"/media/cdrom/linux/tools/ arm-linux-gcc-3.4.5-glibc-2.3.6-linux.tar.bz2"** (from CD) to the folder of Linux system.

The installation command is:

```
sudo tar –jxvf arm-linux-gcc-3.4.5-glibc-2.3.6-linux.tar.bz2 –C /
```

Note: The cross-compiler must need the root permissions for install to /usr/crosstool/gcc-3.4.5-glibc-2.3.6/arm-linux/ directory.

4.2 Setting the Cross-compiler Environment

Define the path of compiler:

```
export PATH=/usr/crosstool/gcc-3.4.5-glibc-2.3.6/arm-linux/bin/:$PATH
```

Note:You can copy it to ".baserc" file, that it'll add the PATH automatically when OS is booting.

4.3 Compiler the System of the SBC6020

4.3.1 Ready to Compiler Files

All components of the system's source is located in "linux/source" directory, you must decompression the files before development.

For example:

```
root@LINUXSERVER:~# mkdir embest
root@LINUXSERVER:~# cd embest/
root@LINUXSERVER:~/embest#
      cp /media/cdrom/CD/bootloader/Bootstrap-v1.14.tar.bz2 ./
root@LINUXSERVER:~/embest#
```

```
    cp /media/cdrom/CD/bootloader/u-boot-1.3.4.tar.bz2 ./
root@LINUXSERVER:~/embest# cp /media/cdrom/CD/kernel/linux-2.6.30.tar.bz2 ./
root@LINUXSERVER:~/embest#
    cp /media/cdrom/CD/filesystem/rfs_SBC6020.tar.bz2 ./
root@LINUXSERVER:~/embest# mkdir tools
root@LINUXSERVER:~/embest# cd tools/
root@LINUXSERVER:~/embest/tools#
cp /media/cdrom/CD/filesystem/mkyaffs2image ./
root@LINUXSERVER:~/embest/tools# cp /media/cdrom/CD/tools/mkimage ./
root@LINUXSERVER:~/embest/tools# export PATH=/root/embest/tools:$PATH
root@LINUXSERVER:~/embest/tools# cd ../
root@LINUXSERVER:~/embest# tar xjf Bootstrap-v1.14.tar.bz2
root@LINUXSERVER:~/embest# tar xjf u-boot-1.3.4.tar.bz2
root@LINUXSERVER:~/embest# tar xjf linux-2.6.30.tar.bz2
root@LINUXSERVER:~/embest# tar xjf rfs_SBC6020.tar.bz2
```

When operation for complete, that you can see linux-2.6.30, u-boot-1.3.4, Bootstrap-v1.14, rfs, tools four directory will be generated under the current folder.

4.3.2 The First Booting Stage -AT91Bootstarp

SBC6020 development board is support for boot from nandflash norflash, commands will be generated nandflash_at91sam9g20ek.bin binary file.

```
root@LINUXSERVER:~/embest# cd Bootstrap-v1.14
root@LINUXSERVER:~/embest/Bootstrap-v1.14# cd board/at91sam9g20ek/nandflash/
root@LINUXSERVER:~/embest/Bootstrap-v1.14/board/at91sam9g20ek/nandflash# make
```

When operation for complete,we need nandflash_at91sam9g20ek.bin binary file in the current directory.

4.3.3 The Second Booting Stage-U-boot

U-boot is used in the second booting stage of SBC6020, enter the following commands in the terminal, u-boot.bin will be generated under current folder.

```
root@LINUXSERVER:~/embest/u-boot-1.3.4#
        make at91sam9g20ek_nandflash_config
root@LINUXSERVER:~/embest/u-boot-1.3.4# make
```

When operation for complete,we need u-boot.bin binary file in the current directory.

4.3.4 The Third Booting Stage-Kernel

You can use the follow commands to complier kernel:

```
root@LINUXSERVER:~/embest/linux-2.6.30#
    cp arch/arm/configs/sbc6020_defconfig .config
```

```
root@LINUXSERVER:~/embest/linux-2.6.30# make menuconfig
root@LINUXSERVER:~/embest/linux-2.6.30# make uImage
```

Wait the compilation of uImage to be finished, and found it under /arch/arm/boot/ folder.

4.3.5 Make Root File System Image

Use the follow commands to generated rtfs.yaffs2 file as we need.

```
root@LINUXSERVER:~/embest# mkyaffs2image rfs/ rfs.yaffs2
```

4.4 System Customization

In fact, there are many kernel configuration options at Linux kernel, users can Increase or reduction of kernel features, make better suited to the needs of users. The next chapter is an example for kernel configuration process.

4.4.1 Modify the Kernel Configuration

Kernel source code in the factory provided the default configuration file:
arch/arm/configs/sbc6020_defconfig
Users can custom system base on the default configuration file.

```
root@LINUXSERVER:~/embest/linux-2.6.30#
      cp arch/arm/configs/sbc6020_defconfig .config
root@LINUXSERVER:~/embest/linux-2.6.30# make menuconfig
```

The following configure for each driver Help:

RTC Drive

1.Select Device drivers
2.Select Real Time Clock
3.Select AT91SAM9x

EEPROM Drive

1.Select Device Drivers
2.Select Misc devices
3.Select EEPROM support
4.Select I2C EEPROMs from most vendors

CF card Drive

1.Select Device Drivers
2.Select Serial ATA (prod) and Parallel ATA (experimental) drivers
3.Select ATA SFF support
4.Select PATA support for AT91SAM9260

DM9000 Drive

1.Select Device Drivers
2.Select Network device support

3.Select Ethernet (10 or 100Mbit)

4.Select DM9000 support

MACB Drive

1.Select Device Drivers

2.Select Network device support

3.Select Ethernet(10 or 100Mbit)

4.Select Atmel MACB support

Save the config,compile the kernel by follow command:

```
root@LINUXSERVER:~/embest/linux-2.6.30# make uImage
```

4.4.2 Complier Driver Modules

There has a port of SBC6020 development board is compiler by modular approach, that under CD's drivers folder. Complier driver needs kernel. So you need complier kernel before that complier driver. In addition the need to pay attention to path of the kernel with the kernel driver module path is consistent.

This is GPIO module driver's Makefile content.

```
 ifneq ($(KERNELRELEASE),)
      obj-m := gpio_dev.o
else
      KERNELDIR ?= ../../kernel/linux-2.6.30

      PWD := $(shell pwd)
all:
      $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
      rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions Module.symvers
endif
```

*Note: KERNELDIR ?= **../../kernel/linux-2.6.30** must be set correct, if not that cannot completion of the compiler.*

Modify Makefile file about kernel's position, for example:

```
KERNELDIR ?= ../linux-2.6.30
```

Use following command to complier GPIO driver.

```
root@LINUXSERVER:~/embest/gpio_dev# make
```

Wait for the compilation of complier, then gpio_dev.ko will be generated under the current folder. Download that file to target board and use insmod command to load the module.

You can downloader the object driver to the target board by network.You should connet your board to the PC by net cable. And then set the board IP address and open the software tftpd service.Follow the command:

```
tftp -r gpio_dev.ko -g 192.192.192.105
```

-r: The name of the driver that you want to downloader to your target board.

-g: The IP address of the tftpd service.

Affer ttat ,The driver will downloader to your target board, and you can usr the insmod command to load the driver to system.

Chapter 5 Update the System Image

The development board with three types of memory respectively Nandflash, currently supports two types of memory from startup.

5.1 System Image Map

5.1.1 The Map of Nandflash Startup

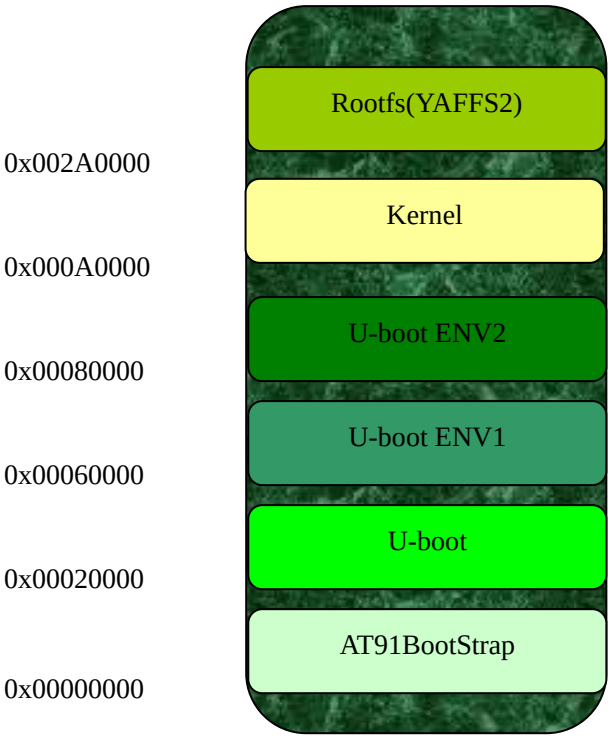

Figure 5.1.1 Nandflash image map

5.2 Setting the Environment of Burning Image

Install the application *02 Linux2.6 Kit\02 Tools\Install AT91-ISP v1.12.exe* (or higher) in your PC. You will see the icon *SAM-BA v2.8* appearing on the desktop of PC.

5.3 Burning System Image

5.3.1 Burning the Image for Nandflash Startup

- Connect debug serial port. About hardware connects can refer to "Hardware manual of SBC6020" .
- Connect USB to your PC using USB cable.
- Open the HyperTerminalon the PC,config the baud 115200, 8 bit data, 1 bit stop, none parity,none Flow control.
- Before power on,open the two jumper cap(J8 and J11). Then pown on,the HyperTerminalon will output follows

```
RomBOOT
>
```

- Double click the ***SAMBA v2.8*** cursor on the desktop of the PC.

Figure 5.3.1.1 SAM-BA v2.8 sofeware icon

- Shown in the figure below, select "***AT91SAM9g20-EK***" and "***\usb\ARM0***", and click "***Connect***".

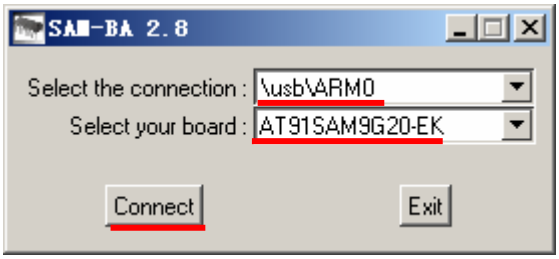

Figure 5.3.1.2 SAM-BA startup interface

- After click "***Connect***", then entering SAM-BA software.

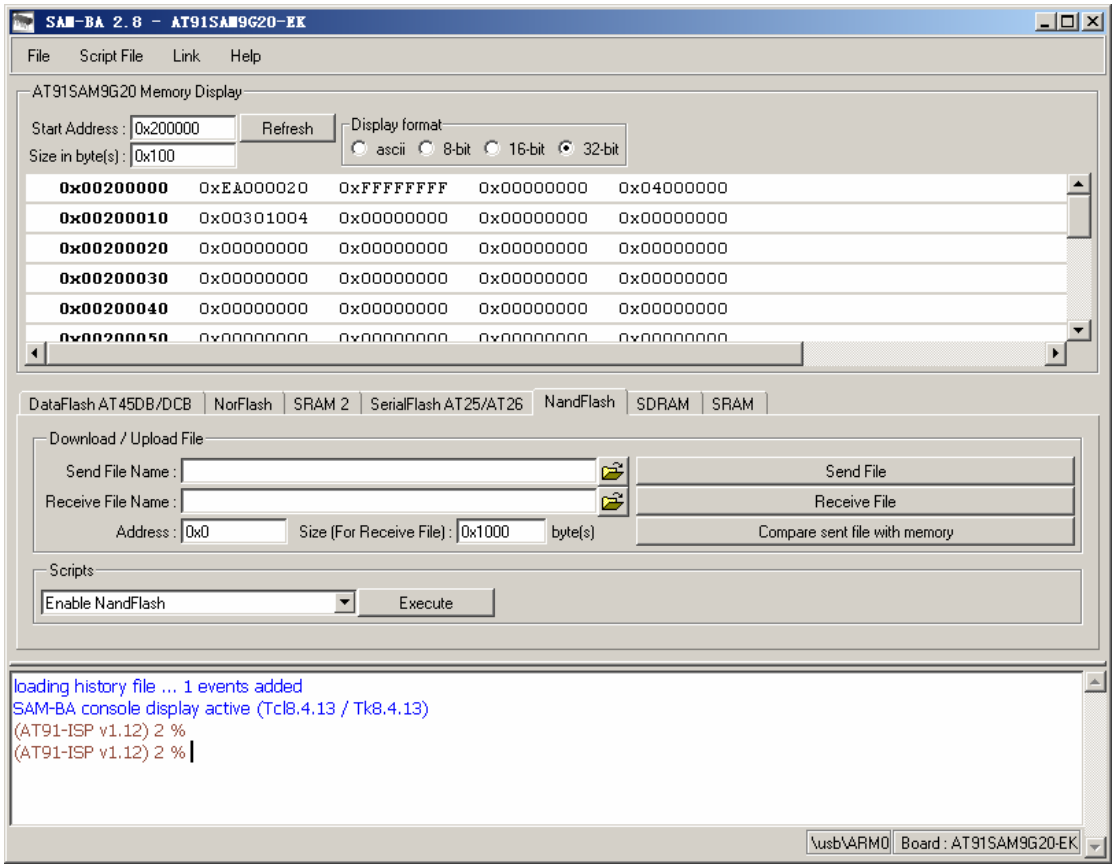

Figure 5.3.1.3 SAM-BA burn interface

- Close the jumper cap(J8 and J11),and Select "***NandFlash***" tab. Then select the options "***Enable NandFlash***" at "***Scripts***" Pull-down menus, then click "***Execute***". Shown in the figure below:

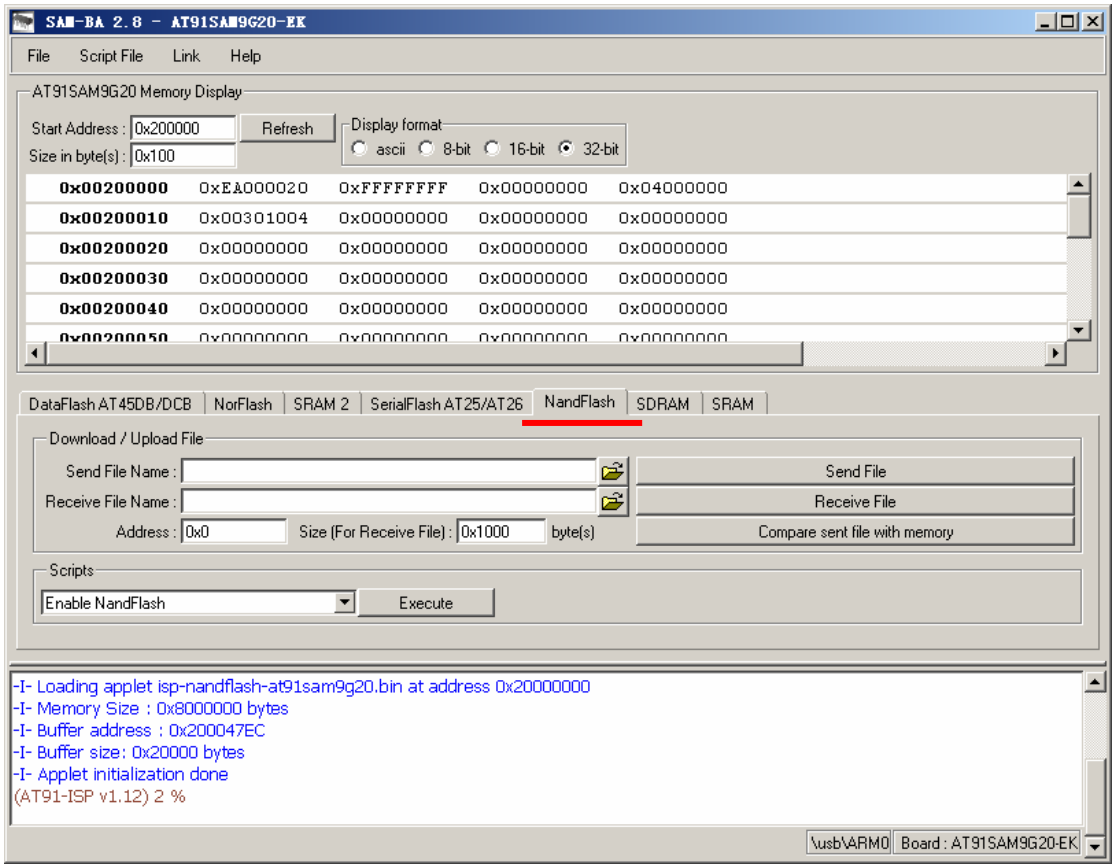

Figure 5.3.1.4 SAM-BA enable nandflash

- At "***Scripts***" Pull-down menus, select "***Erase All***", and click "***Execute***", wait for a moment, you can see nandflash will be Erased.

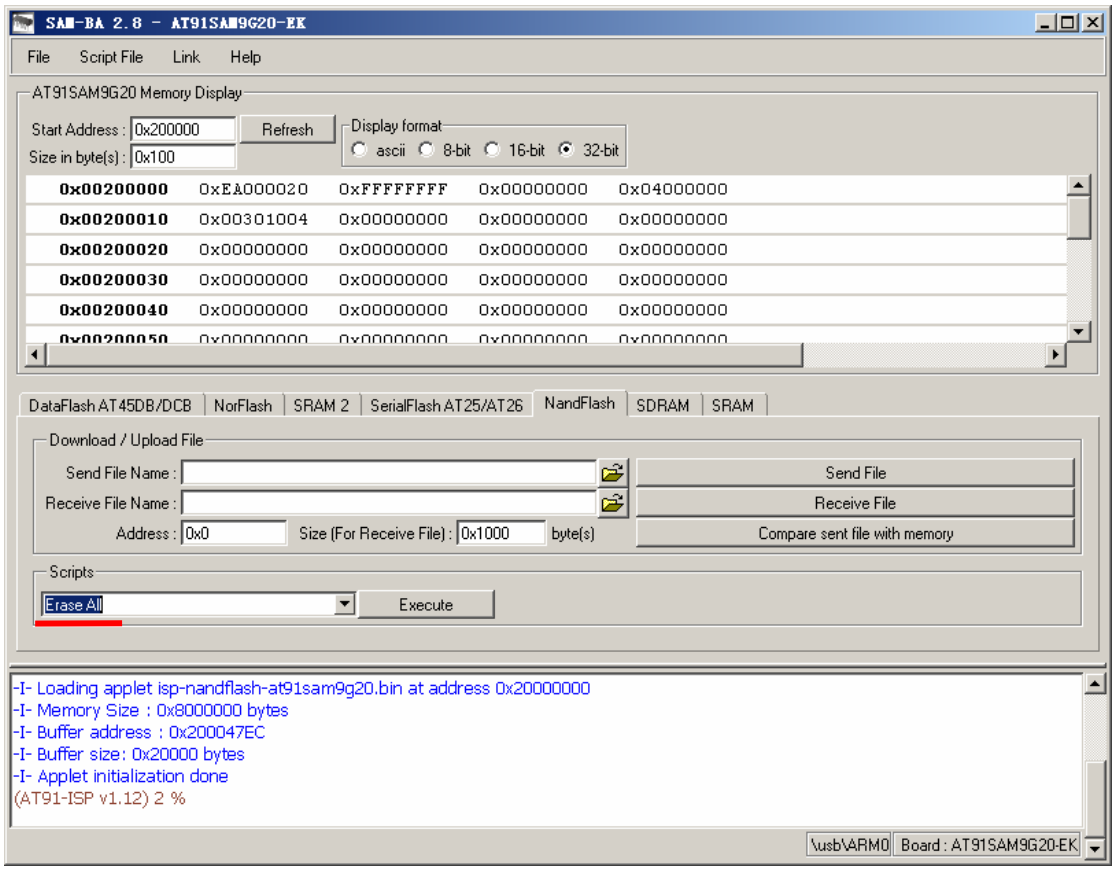

Figure 5.3.1.5 SAM-BA erase nandflash

- At "***Scripts***" Pull-down menus, select "***Scrub NandFlash***", and click "***Execute***", wait for a moment, you can see nandflash will be Erased.

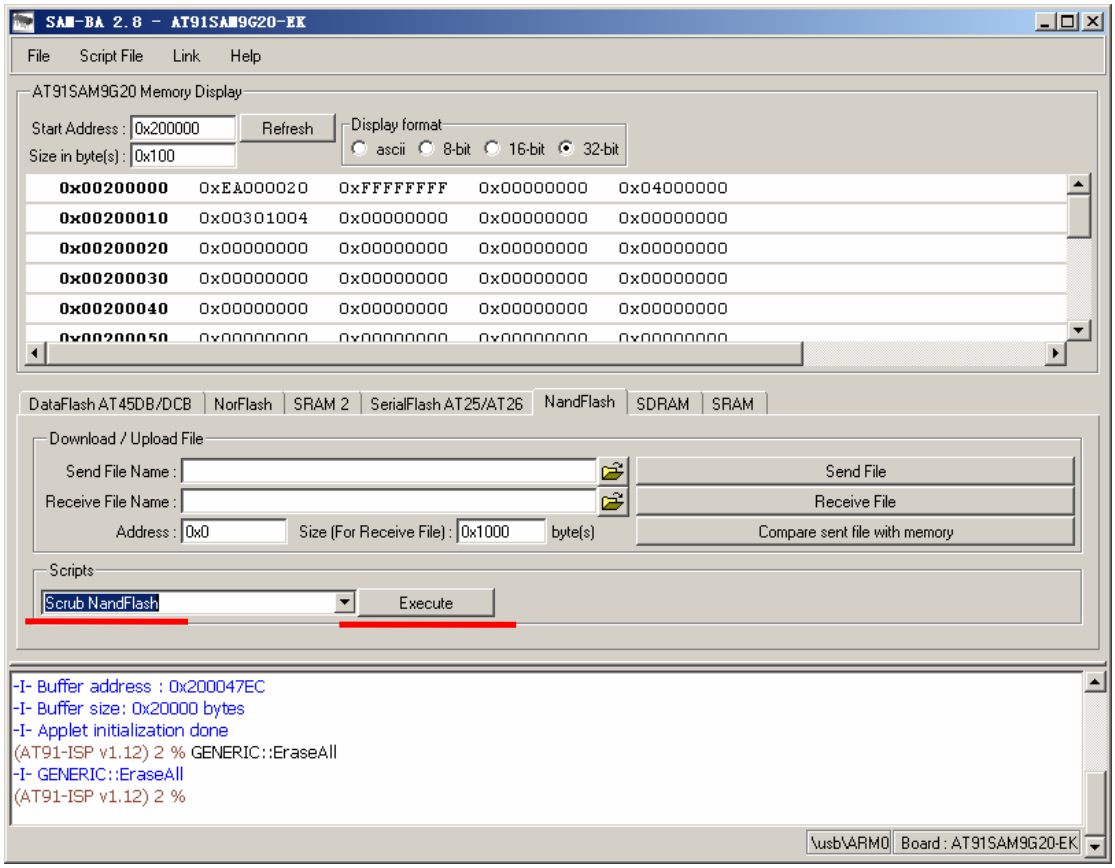

Figure 5.3.1.6 SAM-BA erase nandflash

- At "***Scripts***" Pull-down menus, select "***Send Boot File***", and click "***Execute***"

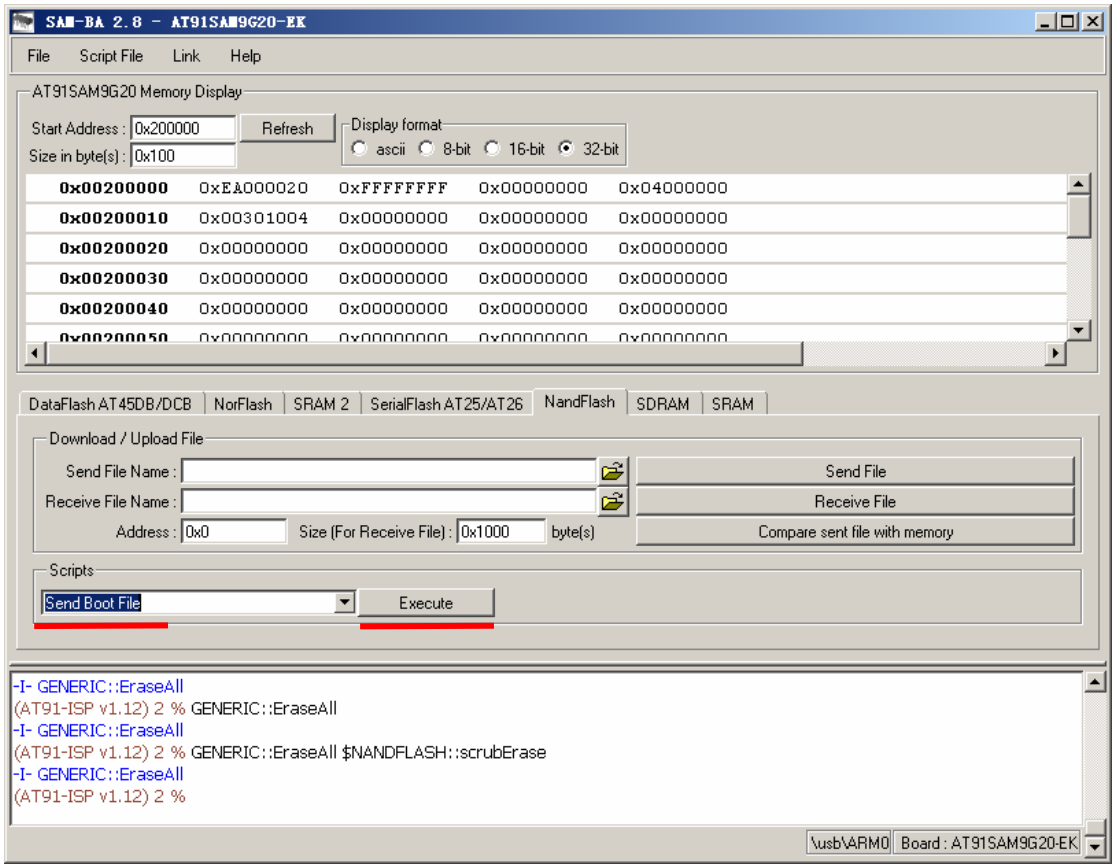

Figure 5.3.1.7 SAM-BA burn nandflash_at91sam9g20ek.bin

- When you click "***Execute***", you will open a dialog box, select "***nand_at91sam9g20ek.bin***" and click open. Then you can download this file to target board.

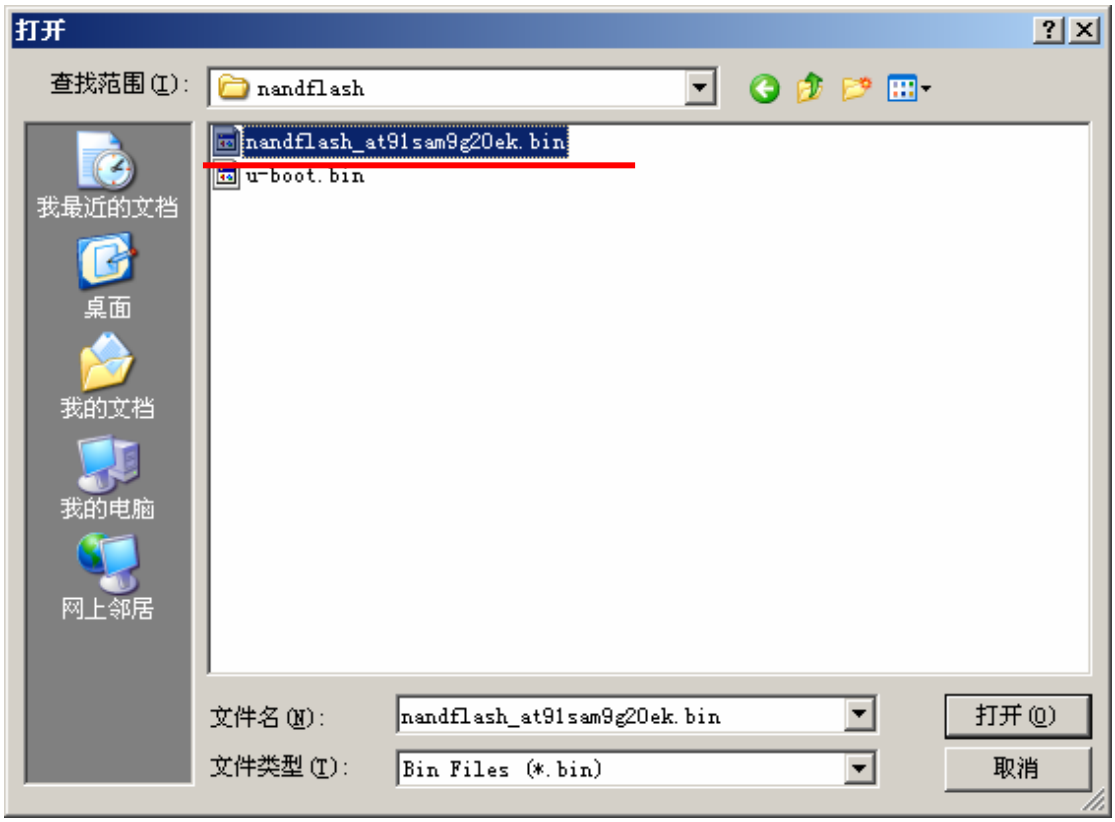

Figure 5.3.1.8 SAM-BA burn nand_at91sam9g20ek.bin

- Enter the downloader address of U-boot in the "***address***" field: "***0x20000***"

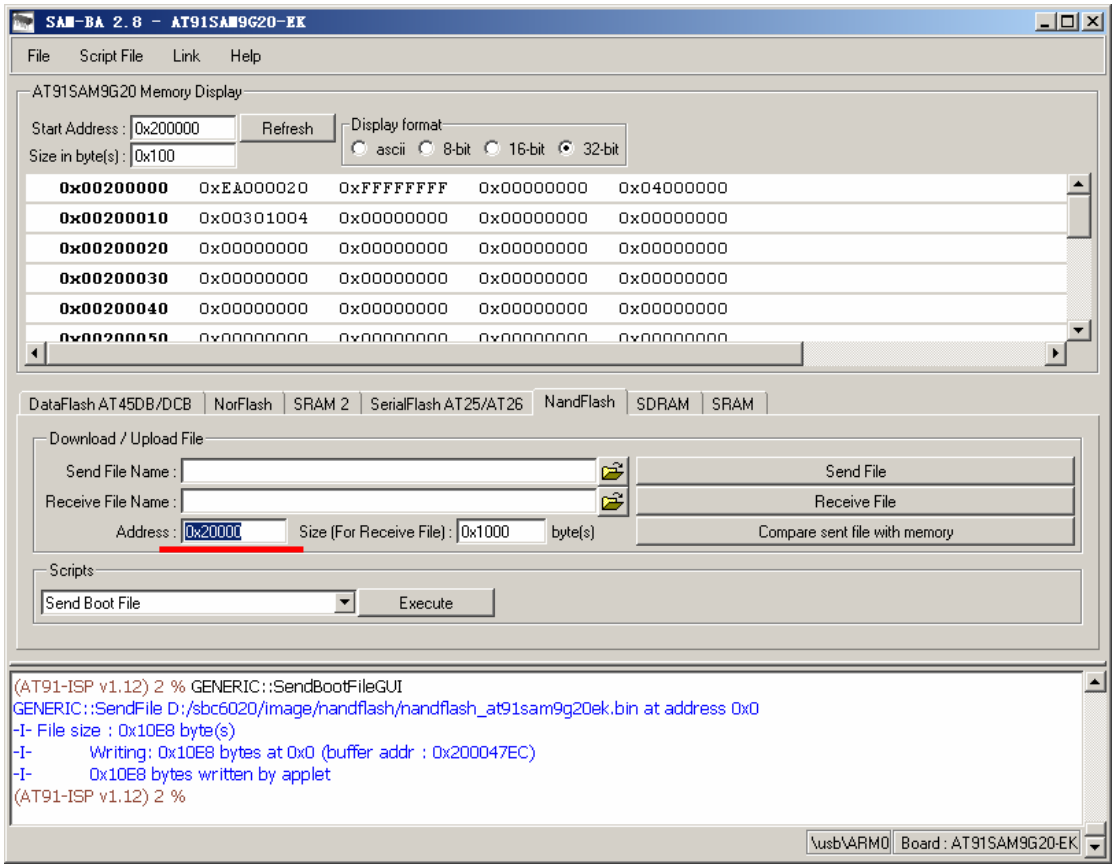

Figure 5.3.1.9 SAM-Ba burn u-boot.bin

- Open the "***Send File Name***" browse window, click "***open folder***" button.

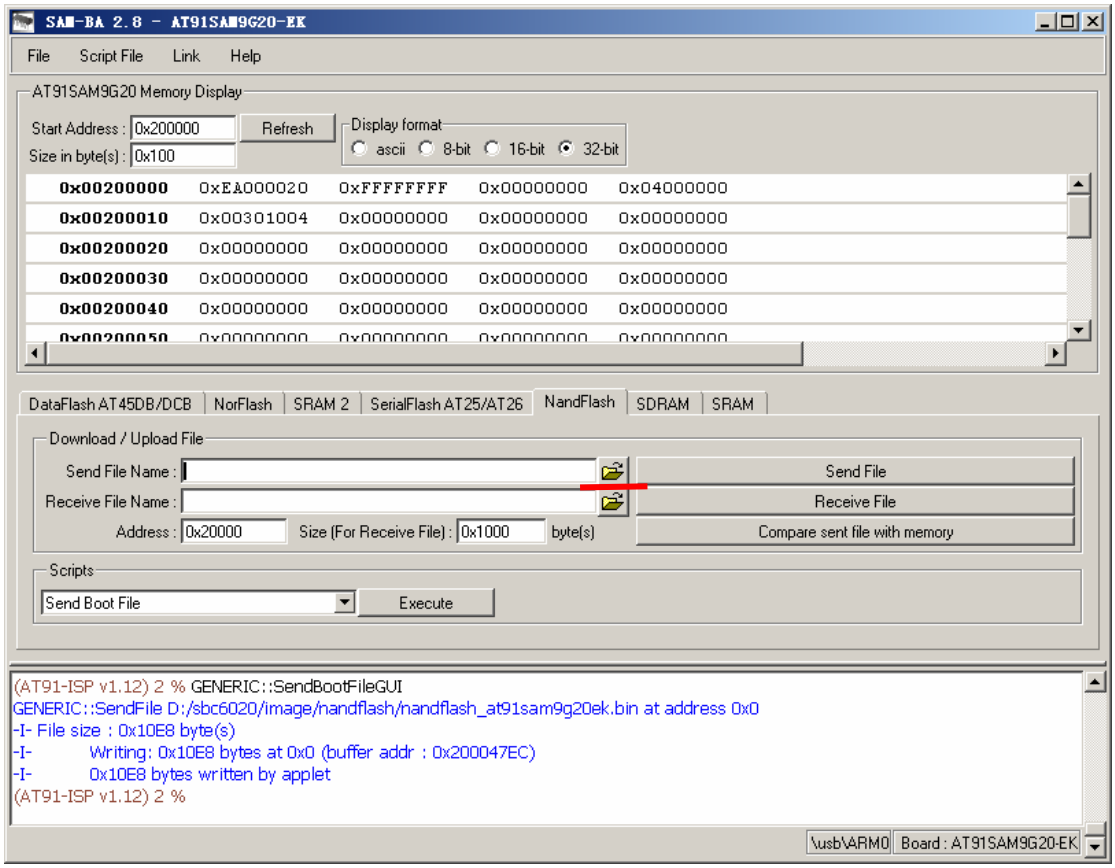

Figure 5.3.1.10 SAM-BA burn u-boot.bin

➢ You will open a dialog box, select "***u-boot.bin***" and click open.

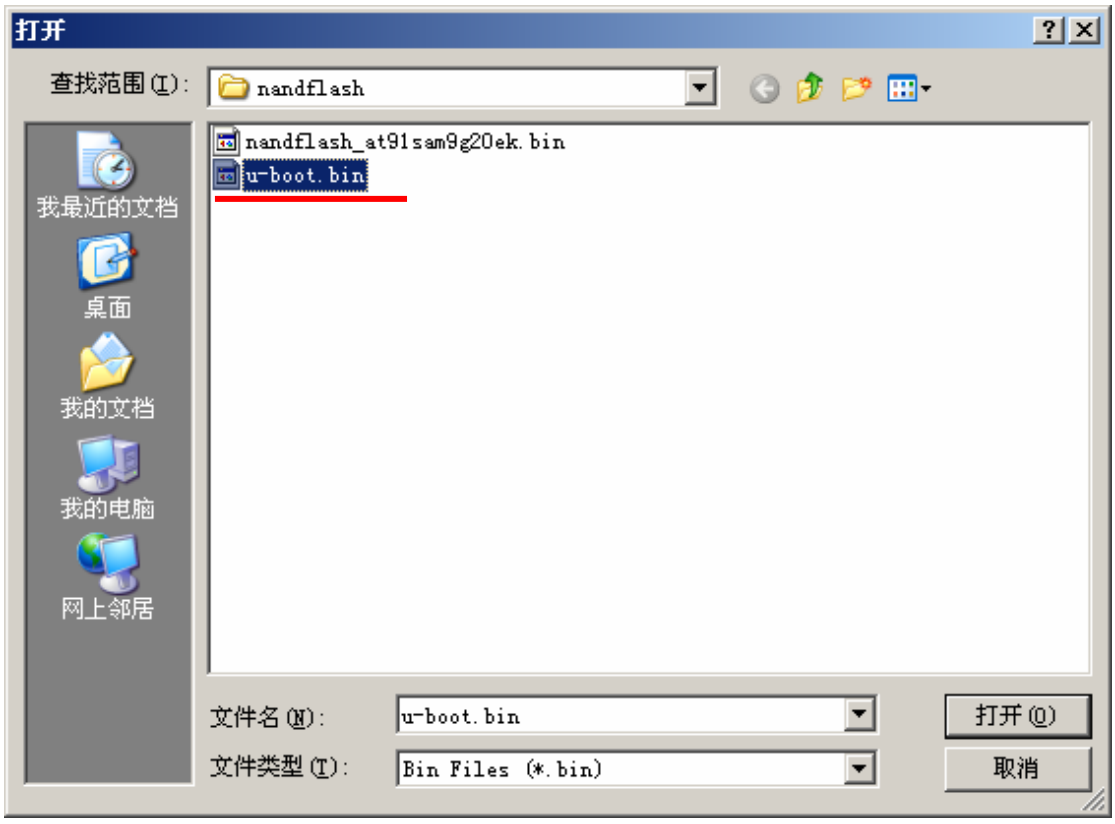

Figure 5.3.1.11 SAM-BA burn u-boot.bin

➢ Then download "***u-boot.bin***" to target board nandflash, click "***Send File***" button

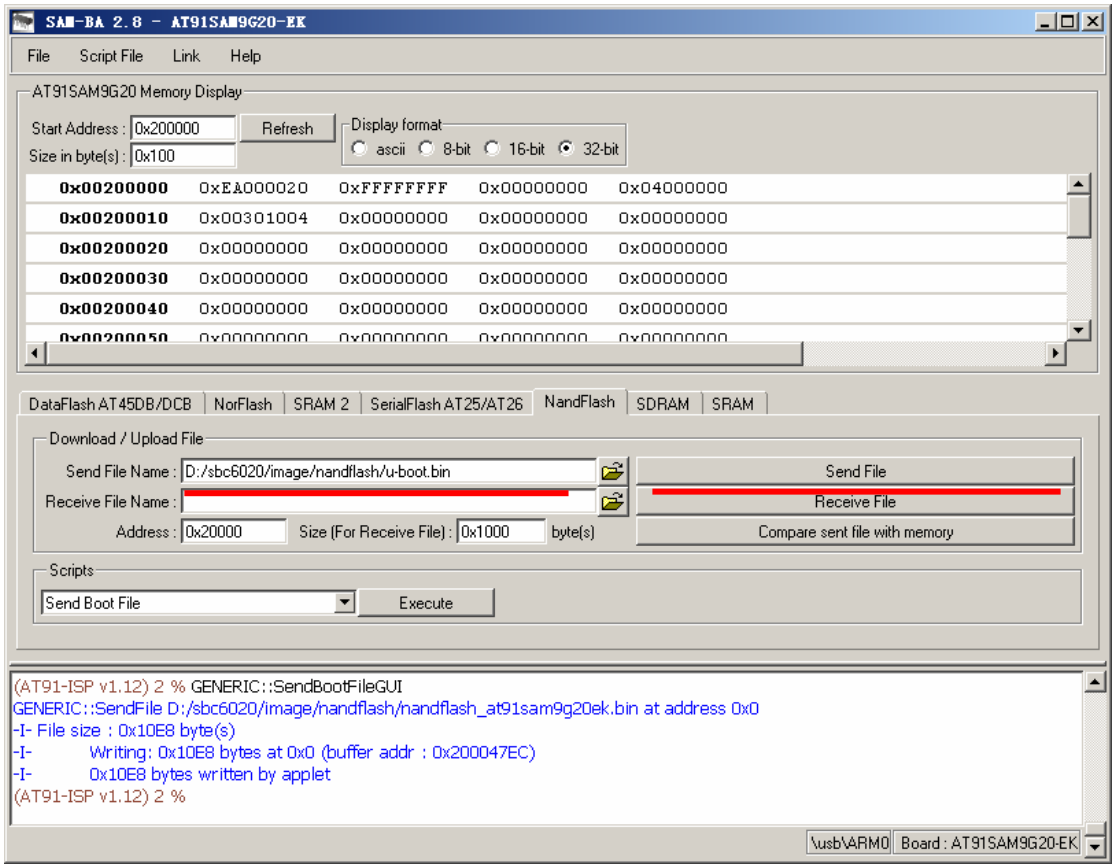

Figure 5.3.1.12 SAM-Ba burn u-boot.bin

➢ Enter the downloader address of U-boot in the "***address***" field: "***0xA0000***"

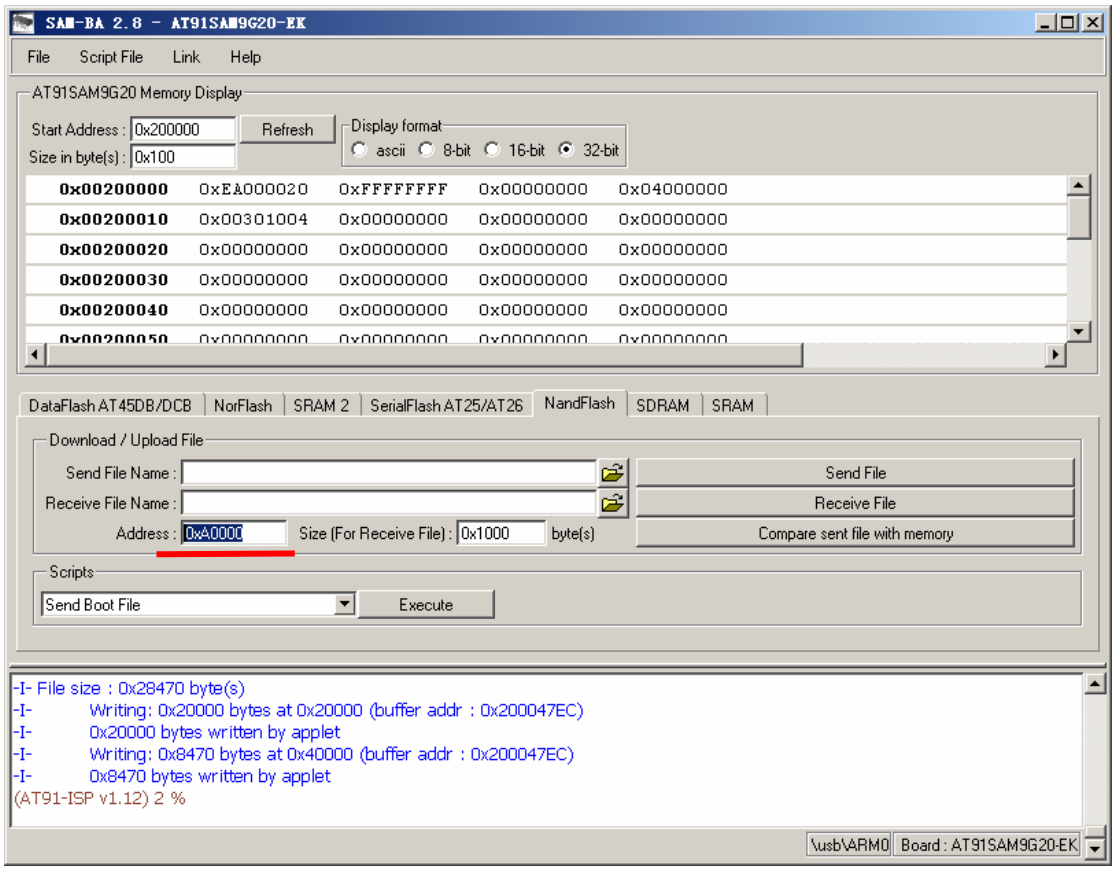

Figure 5.3.1.13 SAM-BA burn uImage

- Open the "***Send File Name***" browse window, click "***open folder***" button.

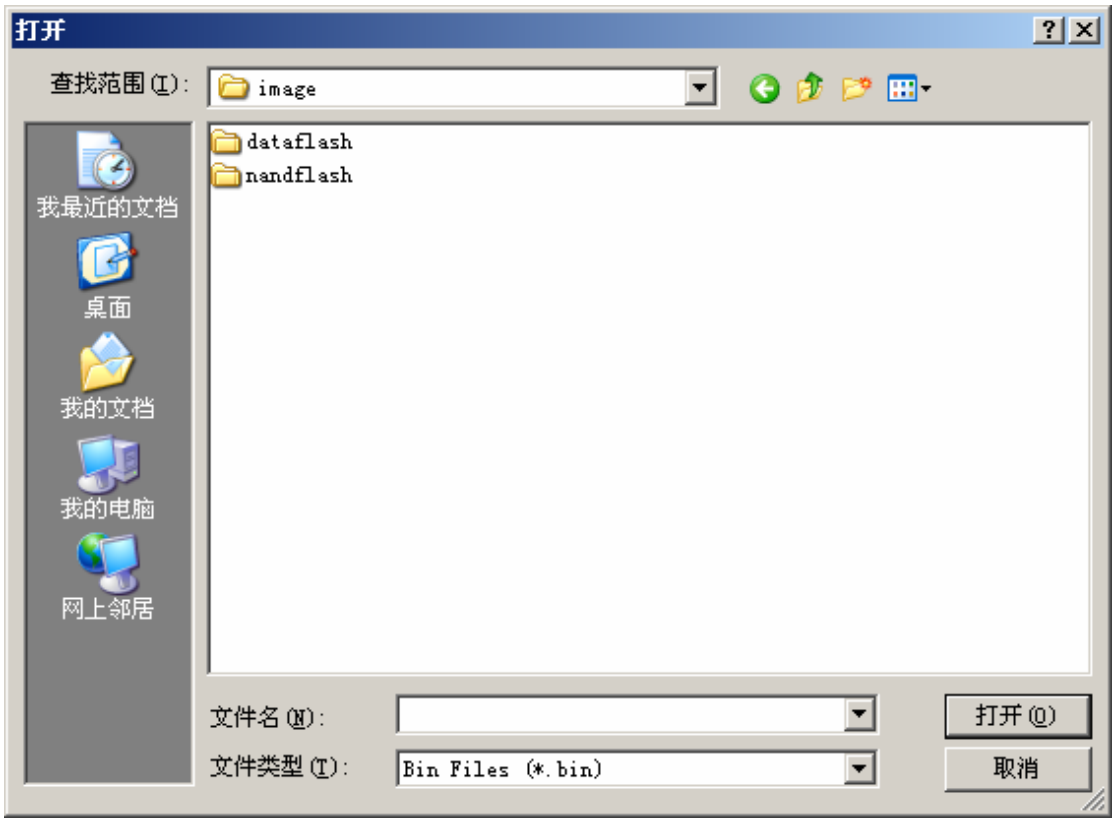

Figure 5.3.1.14 SAM-BA burn u-boot.bin

- Open the "***Send File Name***" browse window, then you should select "***All Files (*.*)***" in the "***files of type***", select "***uImage***", as shown in the figure below.

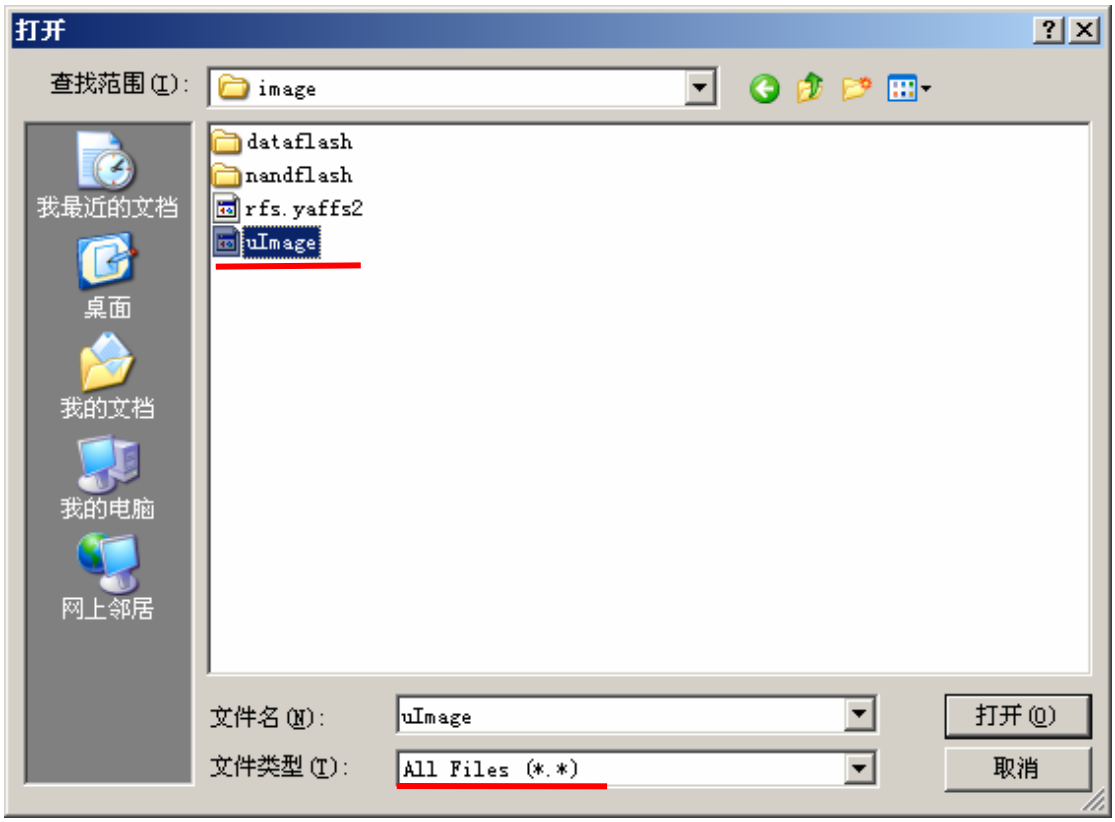

Figure 5.3.1.15 SAM-BA burn uImage to nandflash

- Then download "***uImage***" to target board nandflash, click "***Send File***" button.

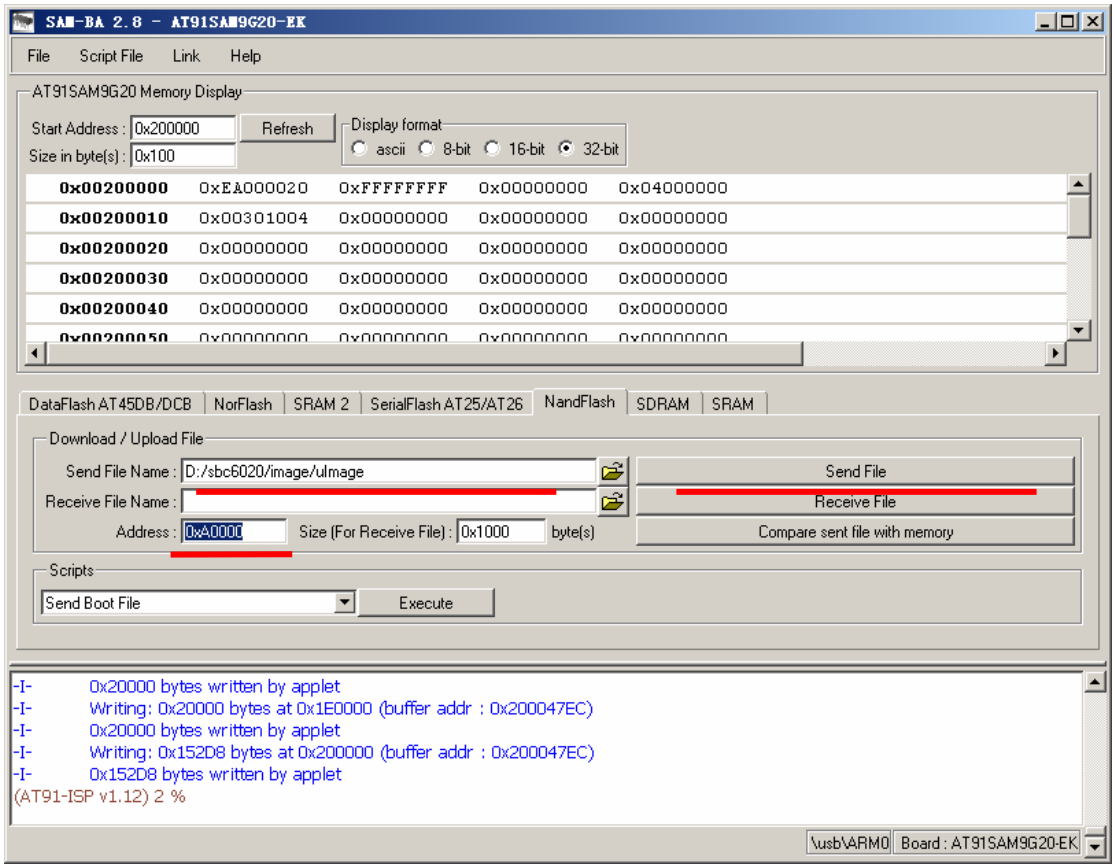

Figure 5.3.1.16 SAM-BA burn uImage to nandflash

- Make target board connect to network.
- Set tftpd server and download folder. Copy CD's tools folder to your PC.

Figure 5.3.1.17 Tftpd.exe software icon

- Double click "***tftpd.exe*** ", the following picture will appear on the screen.

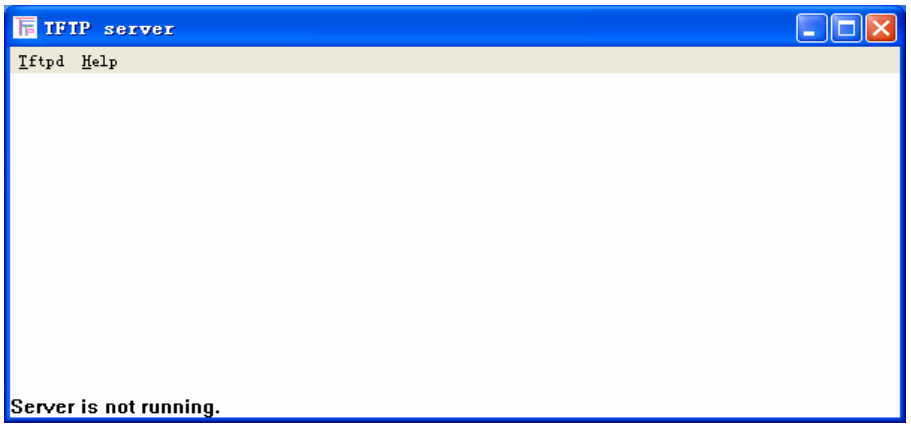

Figure 5.3.1.18 Tftpd.exe open interface

- Click "***Tftpd*** ", and click the option "***Configure*** " in its pulldown menu.

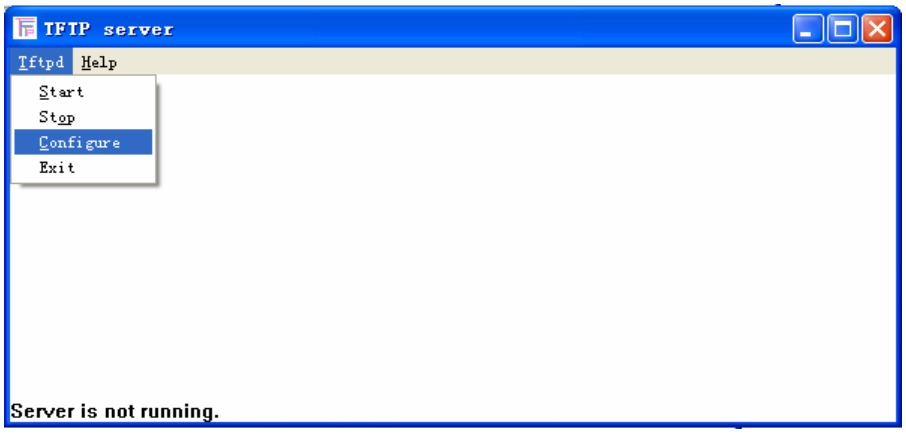

Figure 5.3.1.19 Tftpd.exe config interface

- Press the button ***Browse***, the following picture will appear on the screen, select "image record", and click "***OK*** " button.

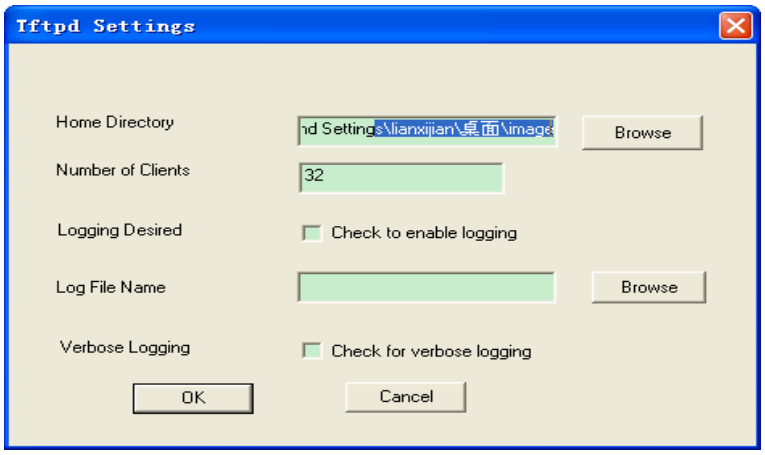

Figure 5.3.1.20 Tftpd.exe config interface

Figure 5.3.1.21 Tftpd.exe config interface

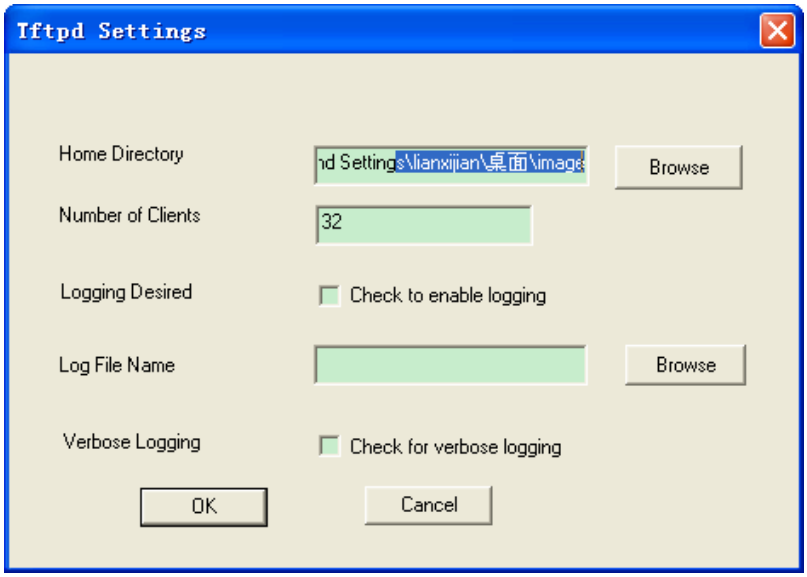

Figure 5.3.1.22 Tftpd.exe config interface

- Click "***Tftpd***", and click the option "***Start***" in its pull-down menu.

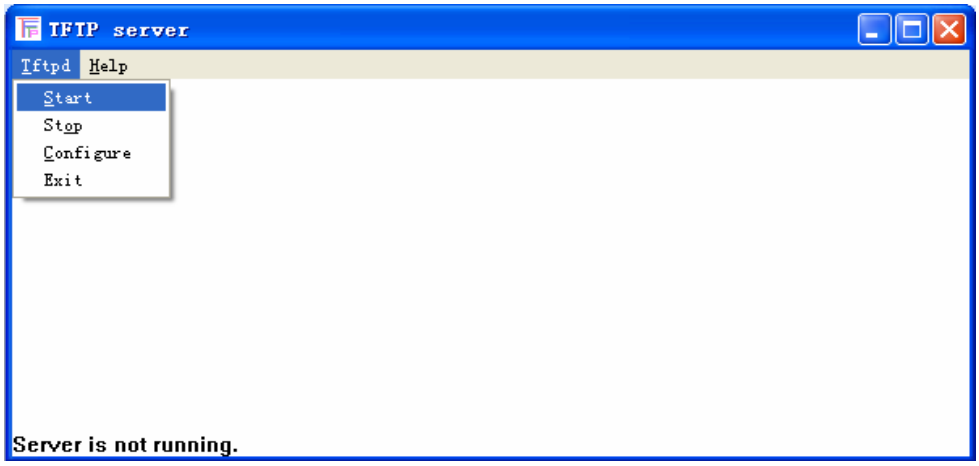

Figure 5.3.1.23 start Tftpd service

- The tftpd server set OK

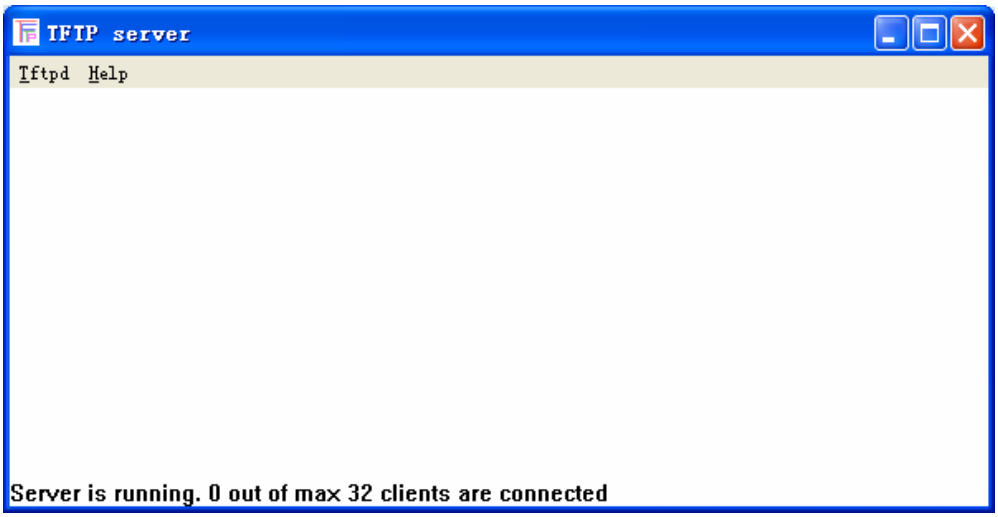

Figure 5.3.1.24 Tftpd service running

- Turn on the target board power, when the U-Boot enter the countdown, press the space key of the keyboard. Then U-Boot enter download image mode.

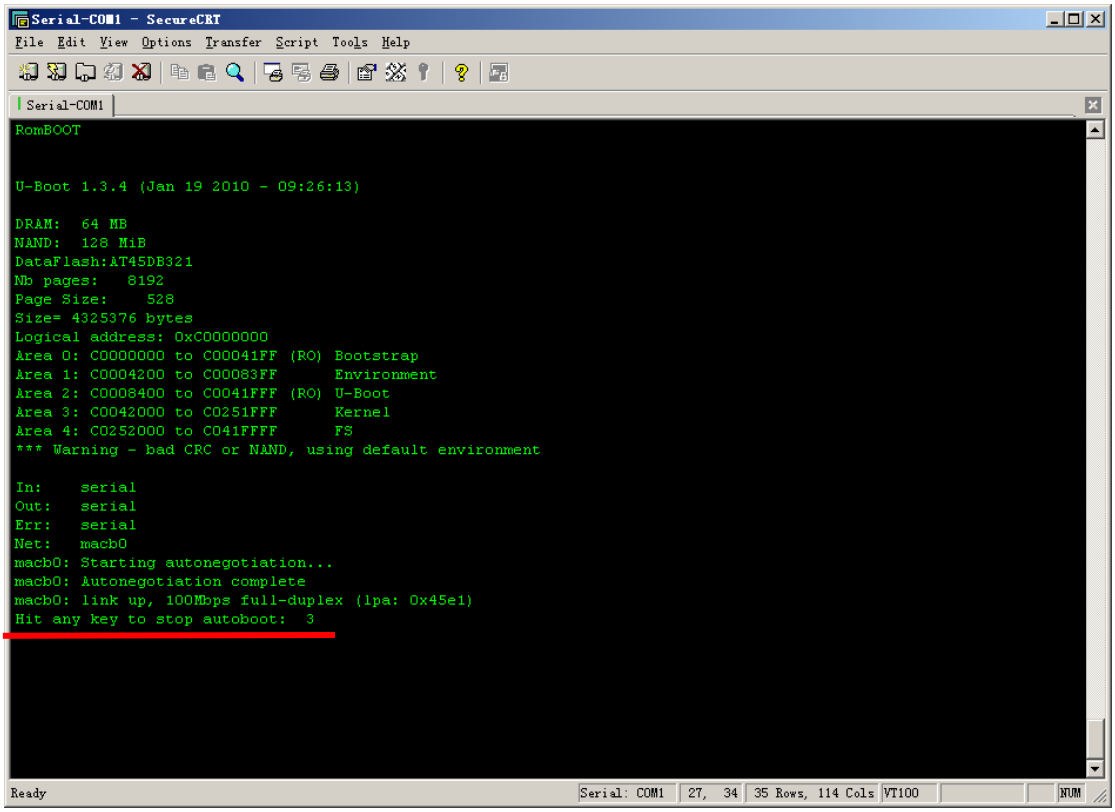
Serial-COM1 - SecureCRT
File Edit View Options Transfer Script Tools Help
Serial-COM1
RomBOOT
U-Boot 1.3.4 (Jan 19 2010 - 09:26:13)
DRAM: 64 MB
NAND: 128 MiB
DataFlash:AT45DB321
Nb pages: 8192
Page Size: 528
Size= 4325376 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C00041FF (RO) Bootstrap
Area 1: C0004200 to C00083FF Environment
Area 2: C0008400 to C0041FFF (RO) U-Boot
Area 3: C0042000 to C0251FFF Kernel
Area 4: C0252000 to C041FFFF FS
*** Warning - bad CRC or NAND, using default environment
In: serial
Out: serial
Err: serial
Net: macb0
macb0: Starting autonegotiation...
macb0: Autonegotiation complete
macb0: link up, 100Mbps full-duplex (lpa: 0x45e1)
Hit any key to stop autoboot: 3
Ready
Serial: COM1 27, 34 35 Rows, 114 Cols VT100
NUM

Figure 5.3.1.25 U-boot start interface

- Set the target board IP and server IP, gateway IP in the U-boot

```
set serverip 192.192.192.155
set ipaddr 192.192.192.200
saveenv
```

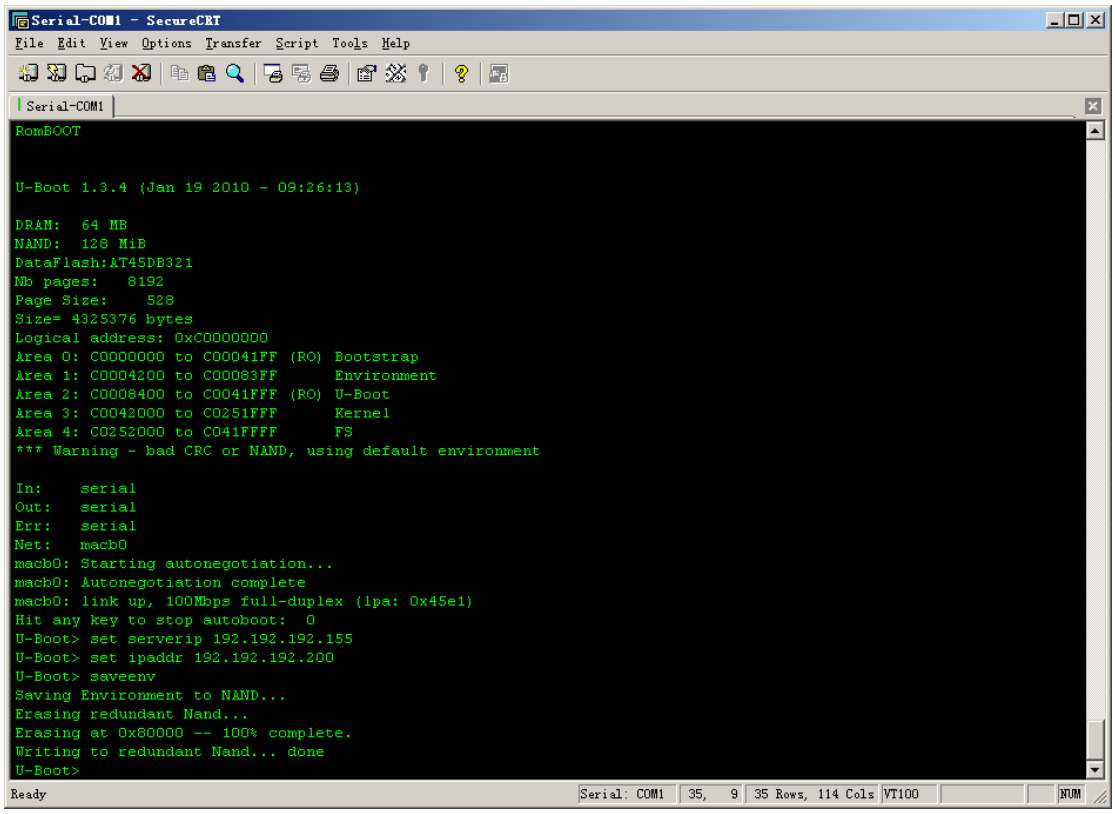
Serial-COM1 - SecureCRT
File Edit View Options Transfer Script Tools Help
Serial-COM1
RomBOOT
U-Boot 1.3.4 (Jan 19 2010 - 09:26:13)
DRAM: 64 MB
NAND: 128 MiB
DataFlash:AT45DB321
Nb pages: 8192
Page Size: 528
Size= 4325376 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C00041FF (RO) Bootstrap
Area 1: C0004200 to C00083FF Environment
Area 2: C0008400 to C0041FFF (RO) U-Boot
Area 3: C0042000 to C0251FFF Kernel
Area 4: C0252000 to C041FFFF FS
*** Warning - bad CRC or NAND, using default environment
In: serial
Out: serial
Err: serial
Net: macb0
macb0: Starting autonegotiation...
macb0: Autonegotiation complete
macb0: link up, 100Mbps full-duplex (lpa: 0x45e1)
Hit any key to stop autoboot: 0
U-Boot> set serverip 192.192.192.155
U-Boot> set ipaddr 192.192.192.200
U-Boot> saveenv
Saving Environment to NAND...
Erasing redundant Nand...
Erasing at 0x80000 -- 100% complete.
Writing to redundant Nand... done
U-Boot>
Ready
Serial: COM1 35, 9 35 Rows, 114 Cols VT100
NUM

Figure 5.3.1.26 U-boot loaddown mode

➢ Downloader rfs.yaffs2 to the target board SDRAM by the TFTP. Follow the command.

tftp 20400000 rfs.yaffs2

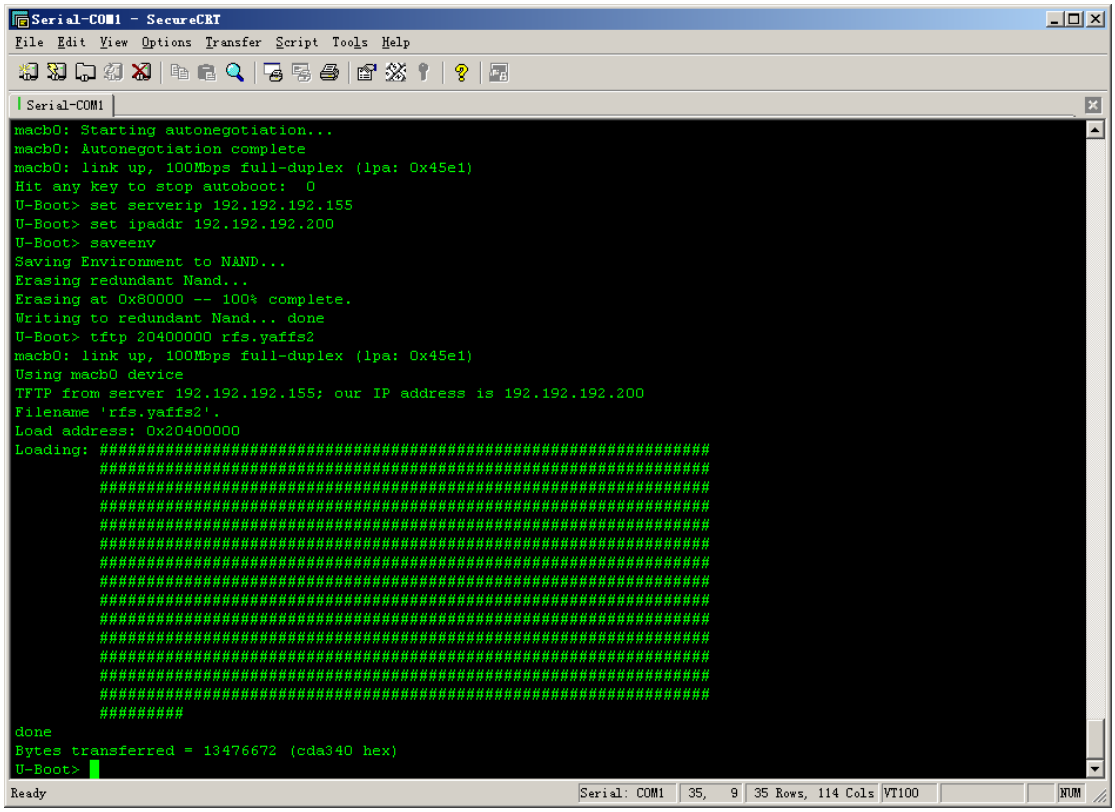
Serial-COM1 - SecureCRT
File Edit View Options Transfer Script Tools Help
Serial-COM1
macb0: Starting autonegotiation...
macb0: Autonegotiation complete
macb0: link up, 100Mbps full-duplex (lpa: 0x45e1)
Hit any key to stop autoboot: 0
U-Boot> set serverip 192.192.192.155
U-Boot> set ipaddr 192.192.192.200
U-Boot> saveenv
Saving Environment to NAND...
Erasing redundant Nand...
Erasing at 0x80000 -- 100% complete.
Writing to redundant Nand... done
U-Boot> tftp 20400000 rfs.yaffs2
macb0: link up, 100Mbps full-duplex (lpa: 0x45e1)
Using macb0 device
TFTP from server 192.192.192.155; our IP address is 192.192.192.200
Filename 'rfs.yaffs2'.
Load address: 0x20400000
Loading: ###
done
Bytes transferred = 13476672 (cda340 hex)
U-Boot>
Ready
Serial: COM1 35, 9 35 Rows, 114 Cols VT100
NUM

Figure 5.3.1.27 update root file system

➢ Erase the nandflash space from 0x2a0000 to the end.

nand erase 0x2A0000

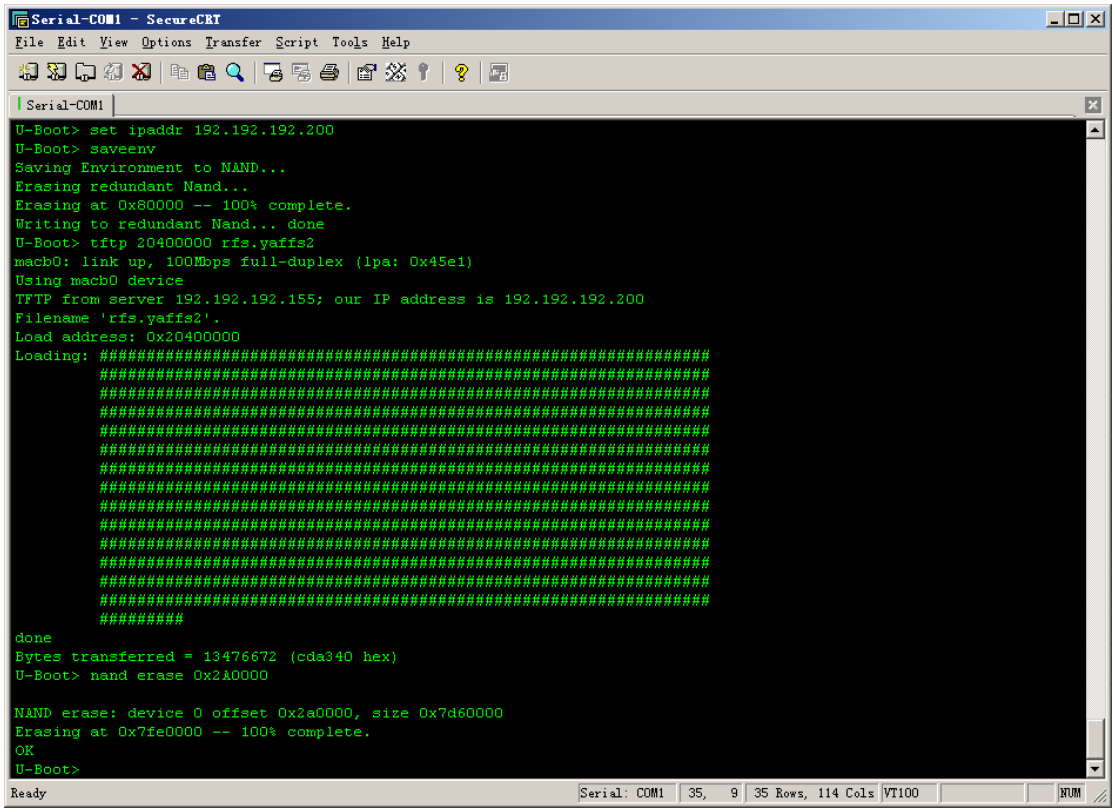
Serial-COM1 - SecureCRT
File Edit View Options Transfer Script Tools Help
Serial-COM1
U-Boot> set ipaddr 192.192.192.200
U-Boot> saveenv
Saving Environment to NAND...
Erasing redundant Nand...
Erasing at 0x80000 -- 100% complete.
Writing to redundant Nand... done
U-Boot> tftp 20400000 rfs.yaffs2
macb0: link up, 100Mbps full-duplex (lpa: 0x45e1)
Using macb0 device
TFTP from server 192.192.192.155; our IP address is 192.192.192.200
Filename 'rfs.yaffs2'.
Load address: 0x20400000
Loading: ##########
done
Bytes transferred = 13476672 (cda340 hex)
U-Boot> nand erase 0x2A0000
NAND erase: device 0 offset 0x2a0000, size 0x7d60000
Erasing at 0x7fe0000 -- 100% complete.
OK
U-Boot>
Ready
Serial: COM1 35, 9 35 Rows, 114 Cols VT100
NUM

Figure 5.3.1.28 Erase nandflash

- Use the nand write.yaffs command to downloader the rfs.yaffs2 to the nandflash.

nand write.yaffs 20400000 0x2A0000 $(filesize)

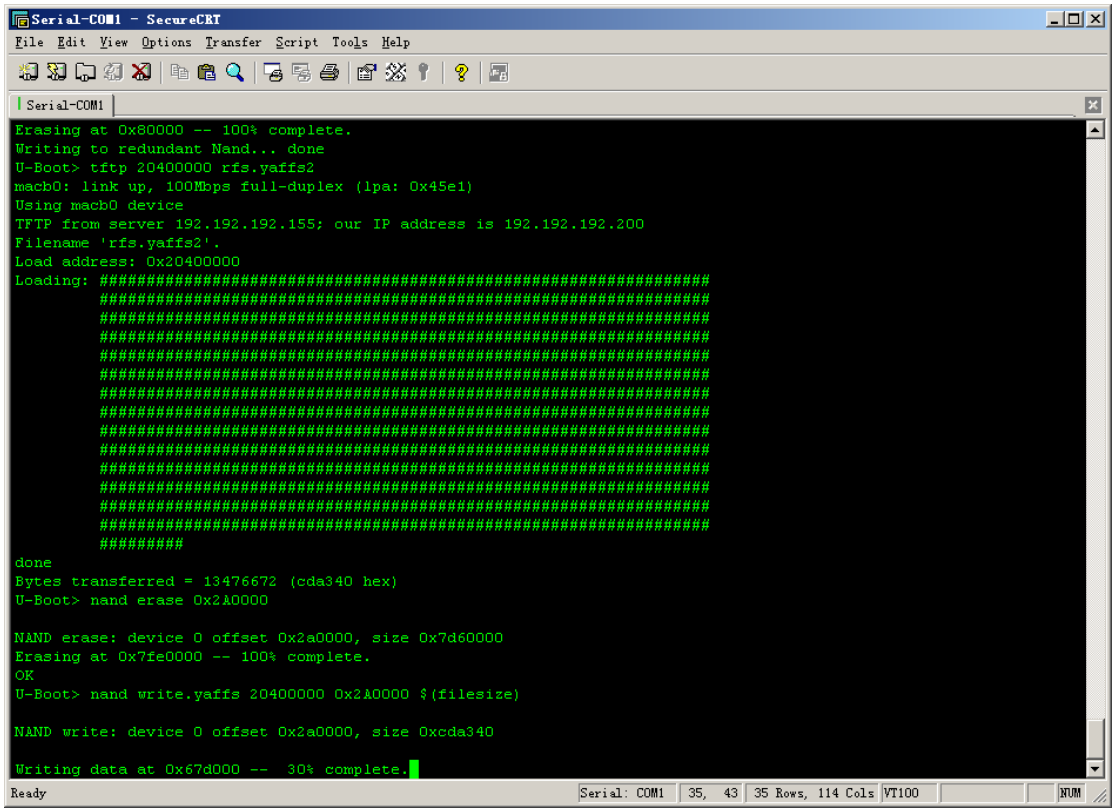
Serial-COM1 - SecureCRT
File Edit View Options Transfer Script Tools Help
Serial-COM1
Erasing at 0x80000 -- 100% complete.
Writing to redundant Nand... done
U-Boot> tftp 20400000 rfs.yaffs2
macb0: link up, 100Mbps full-duplex (lpa: 0x45e1)
Using macb0 device
TFTP from server 192.192.192.155; our IP address is 192.192.192.200
Filename 'rfs.yaffs2'.
Load address: 0x20400000
Loading: ##########
done
Bytes transferred = 13476672 (cda340 hex)
U-Boot> nand erase 0x2A0000
NAND erase: device 0 offset 0x2a0000, size 0x7d60000
Erasing at 0x7fe0000 -- 100% complete.
OK
U-Boot> nand write.yaffs 20400000 0x2A0000 $(filesize)
NAND write: device 0 offset 0x2a0000, size 0xcda340
Writing data at 0x67d000 -- 30% complete.
Ready
Serial: COM1 35, 43 35 Rows, 114 Cols VT100
NUM

Figure 5.3.1.29 Burn root file system to nandflash

➢ Wait for a moment, when the rfs.yaffs2 has writed to nandflash.

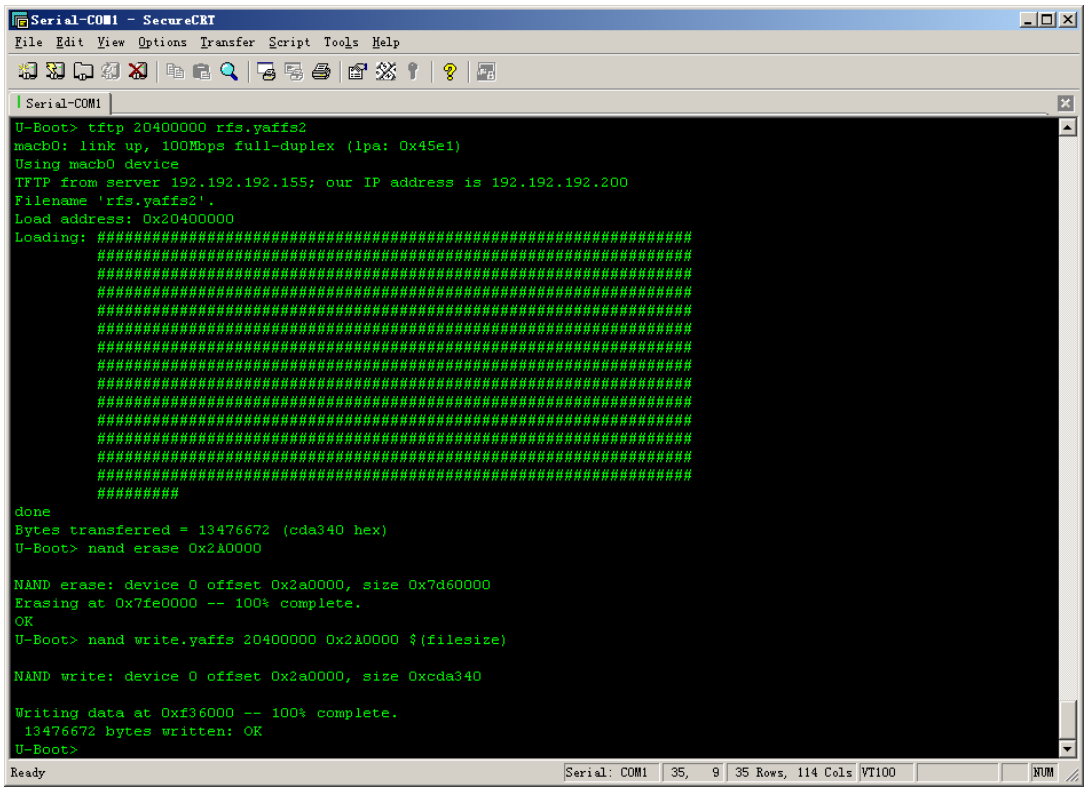

Figure 5.3.1.30 Burn root file system OK

➢ You can use the comand to reset the system:

reset

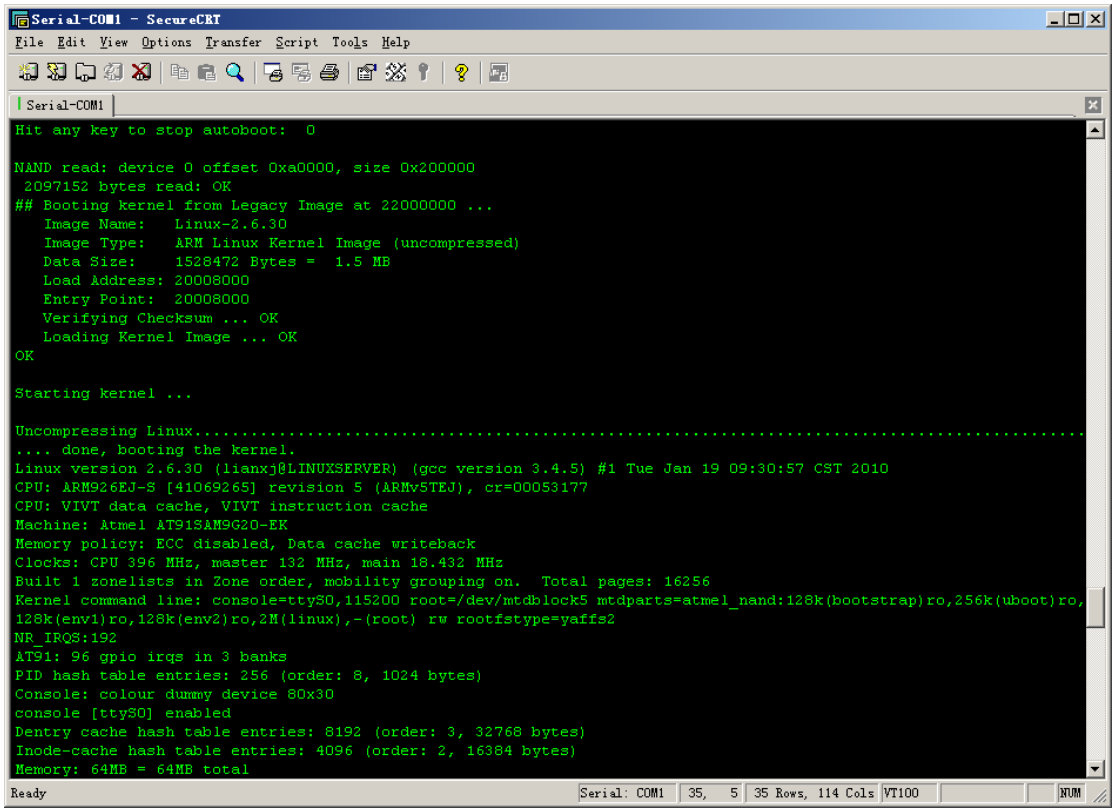
Serial-COM1 - SecureCRT
File Edit View Options Transfer Script Tools Help
Serial-COM1
Hit any key to stop autoboot: 0
NAND read: device 0 offset 0xa0000, size 0x200000
2097152 bytes read: OK
Booting kernel from Legacy Image at 22000000 ...
Image Name: Linux-2.6.30
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 1528472 Bytes = 1.5 MB
Load Address: 20008000
Entry Point: 20008000
Verifying Checksum ... OK
Loading Kernel Image ... OK
OK
Starting kernel ...
Uncompressing Linux.....
.... done, booting the kernel.
Linux version 2.6.30 (lianxj@LINUXSERVER) (gcc version 3.4.5) #1 Tue Jan 19 09:30:57 CST 2010
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00053177
CPU: VIVT data cache, VIVT instruction cache
Machine: Atmel AT91SAM9G20-EK
Memory policy: ECC disabled, Data cache writeback
Clocks: CPU 396 MHz, master 132 MHz, main 18.432 MHz
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 16256
Kernel command line: console=ttyS0,115200 root=/dev/mtdblock5 mtdparts=atmel_nand:128k(bootstrap)ro,256k(uboot)ro,
128k(env1)ro,128k(env2)ro,2M(linux),-(root) rw rootfstype=yaffs2
NR_IRQS:192
AT91: 96 gpio irqs in 3 banks
PID hash table entries: 256 (order: 8, 1024 bytes)
Console: colour dummy device 80x30
console [ttyS0] enabled
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 64MB = 64MB total
Ready
Serial: COM1 35, 5 35 Rows, 114 Cols VT100
NUM

Figure 5.3.1.31 System startup interface

Chapter 6 Application Software Development

6.1 CAN BUS Application Development

SBC6020 provide one-way CAN BUS interface,you can access the device by the "/dev/can0", and provide mainly functions like that refers to table 6-1 content:

Table 6-1 CAN Bus function interface

function	Description
open	Open device
close	Close device
ioctl	Opera CAN controller
read	CAN receive data
write	CAN transmit data

6.1.1 Open Function

This function interfaces are standard Linux file operations function to open the device, the user can use the standard interface to call methods on this function call.

1、Prototype

```
int open(const char *pathname, int flags)
```

2.Return

If the step is successful, the system will return to file descriptor for future use by other function. If the step fails, the system will return to negative value.

3.Parameters

- const char *pathname

Used to transfer device node, where the device node for the (/dev/can0).

- int flags

The file status flags and device access modes of the open device description shall be set according to the value of flags. Mainly includes the following three parameters:

Table 6-2 Open flag

Flag	Description
O_RDONLY	Read only mode
O_WRONLY	Write-only mode
O_RDWR	read/write mode

6.1.2 Close Function

After using the file, if you are not going to use it, you can close the file using “close()”.

1、 Prototype

```
int close(int fd);
```

2.Return

If this step is successful, the system will return to “0”;if the step fails, the system will return to negative value.

3.Parameters

- int fd

User can directly transfer the file descriptor got using open() open function.

6.1.3 Ioctl Function

This function interface is “IO control device” as described in standard linux file operation function. User can dispatch this function using standard ioctl interface dispatch method.

1. Prototype

```
int ioctl(int fd,int cmd,long arg);
```

2. Return value

if this step is successful, the system will return to “0”; if the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- int cmd

Here transfer the operation of the CAN controller, in which specific optional parameters as follows:

CAN_IOCTL_COMMAND:CAN controller to start, stop and reset the request command.

CAN_IOCTL_CONFIG: Some of the properties to configure devices.

Example: acceptance filtering bit timings, mode of the output control register

CAN_IOCTL_STATUS:CAN controller state of the request.

CAN_IOCTL_SEND:a single message over the ioctl interface.

CAN_IOCTL_RECEIVE:Inquiries received information.

CAN_IOCTL_CONFIGURERTR:RTR configuration request automatically

Parameters can view the the file can4linux.h.

- long arg

Transfer the third Parameters is address of structure Parameters, the structure dependent cmd Parameters. Concrete structure has the following several:

- ✓ struct Command_par
- ✓ struct Config_par
- ✓ struct CanStatusPar
- ✓ struct ConfigureRTR_par
- ✓ struct Receive_par
- ✓ struct Send_par

Concrete data structure is following:

✓ struct Command_par structure

```
struct Command_par {
    int cmd;                /**< special driver command */
    int target;             /**< special configuration target */
    unsigned long val1;     /**< 1. parameter for the target */
    unsigned long val2;     /**< 2. parameter for the target */
    int error;              /**< return value */
    unsigned long retval;   /**< return value */
};
```

struct Command_par structure of cmd Parameters configuration

```
# define CMD_START          1
# define CMD_STOP           2
# define CMD_RESET          3
# define CMD_CLEARBUFFERS   4
```

struct Command_par structure of target Parameters configuration

```
# define CONF_ACC           0   /* mask and code */
# define CONF_ACCM          1   /* mask only */
# define CONF_ACCC          2   /* code only */
# define CONF_TIMING        3   /* bit timing */
# define CONF_OMODE         4   /* output control register */
# define CONF_FILTER        5
# define CONF_FENABLE       6
# define CONF_FDISABLE      7
# define CONF_LISTEN_ONLY_MODE  8   /* for SJA1000 PeliCAN */
# define CONF_SELF_RECEPTION    9   /* */
# define CONF_BTR               10  /* set direct bit timing registers
                                    (SJA1000) */
# define CONF_TIMESTAMP         11  /* use TS in received messages */
# define CONF_WAKEUP            12  /* wake up processes */
```

✓ struct CanStatusPar structure

```
IOCTL generic CAN controller status request parameter structure */
typedef struct CanStatusPar {
    unsigned int baud;          /**< actual bit rate */
```

```
        unsigned int status;            /**< CAN controller status register */
        unsigned int error_warning_limit;    /**< the error warning limit */
        unsigned int rx_errors;        /**< content of RX error counter */
        unsigned int tx_errors;        /**< content of TX error counter */
        unsigned int error_code;            /**< content of error code register */
        unsigned int rx_buffer_size;      /**< size of rx buffer   */
        unsigned int rx_buffer_used;      /**< number of messages */
        unsigned int tx_buffer_size;      /**< size of tx buffer   */
        unsigned int tx_buffer_used;      /**< number of messages */
        unsigned long retval;          /**< return value */
        unsigned int type;             /**< CAN controller / driver type */
    } CanStatusPar_t;
```

- ✓ struct ConfigureRTR_par structure

```
    typedef struct ConfigureRTR_par {
        unsigned message;          /**< CAN message ID */
        canmsg_t *Tx;          /**< CAN message struct   */
        int error;             /**< return value for errno */
        unsigned long retval;     /**< return value */
    } ConfigureRTR_par_t ;
```

- ✓ struct Send_par structure

```
    /**
     IOCTL Send request parameter structure */
    typedef struct Send_par {
        canmsg_t *Tx;          /**< CAN message struct   */
        int error;             /**< return value for errno */
        unsigned long retval;     /**< return value */
    } Send_par_t ;
```

- ✓ struct Receive_par structure

```
    /**
     IOCTL Receive request parameter structure */
    typedef struct Receive_par {
        canmsg_t *Rx;          /**< CAN message struct   */
        int error;             /**< return value for errno */
        unsigned long retval;     /**< return value */
    } Receive_par_t ;
```

6.1.4 Read Function

This function interface is “read device” as described in standard linux file operation function. user can dispatch this function using standard interface dispatch method.

1. Prototype

```
    ssize_t read(int fd, char * buf ,size_t count);
```

2. Return

If the step is successful, the system will return to the number of bytes read; if the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- char * buf

Here the definition of transfer for the address of structure canmsg_t, Application to read from the driver to receive the CAN data into the buffer in buf. This is data structure of canmsg_t’s prototype:

```
/**
* The CAN message structure.
* Used for all data transfers between the application and the driver
* using read() or write().
*/
typedef struct {
    /** flags, indicating or controlling special message properties */
    int                 flags;
    int                 cob;        /**< CAN object number, used in Full CAN    */
    unsigned    long id;          /**< CAN message ID, 4 bytes    */
    struct timeval    timestamp;    /**< time stamp for received messages */
    short         int    length;    /**< number of bytes in the CAN message */
    unsigned    char data[CAN_MSG_LENGTH]; /**< data, 0...8 bytes */
} canmsg_t;
```

canmsg_t data structure flag Parameters:

```
#define MSG_ACTIVE    (0)        /**< Controller Error Active */
#define MSG_BASE      (0)        /**< Base Frame Format */
#define MSG_RTR       (1<<0)         /**< RTR Message */
#define MSG_OVR       (1<<1)         /**< CAN controller Msg overflow error */
#define MSG_EXT       (1<<2)         /**< extended message format */
#define MSG_SELF      (1<<3)         /**< message received from own tx */
#define MSG_PASSIVE (1<<4)         /**< controller in error passive */
#define MSG_BUSOFF          (1<<5)          /**< controller Bus Off    */
#define MSG_WARNING         (1<<6)          /**< CAN Warning Level reached */
#define MSG_BOVR      (1<<7)          /**< receive/transmit buffer overflow */
```

canmsg_t data structure CAN message ID Parameters:

```
/* valid bits in CAN ID for frame formats */
#define CAN_SFF_MASK 0x000007FFU      /* standard frame format (SFF) */
#define CAN_EFF_MASK 0x1FFFFFFFU      /* extended frame format (EFF) */
#define CANDRIVERERROR    0xFFFFFFFFul      /* invalid CAN ID == Error */
```

- size_t count

Set the number of bytes that application reads from drive.

6.1.5 Write Function

This function interface is “write device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

1、 Prototype

```
ssize_t write (int fd,const char * buf,size_t count);
```

2. Return

If the step is successful, the system will return to the number of bytes written, If the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- char * buf

Here what is transferred is defined Write initial address of buffer. Application program transmits data of this buffer to drive and sends it out.

- size_t count

Set the number of bytes that application program transmits to serial port.

6.2 UART Application Development

This group of functions provides receiving and transmitting operations of 7 uart devices. They control different uart by transferring different nodes. Table 6-3 shows the names of the nodes to be used in the program and the relations of hardware interfaces. For locations of each interface on SBC6020 board please refer to “Hardware manual of SBC6020”.

Table6-3 Node and hardware interface

Names of nodes	Interface
ttyS0	COM0 (DEBUG)
ttyS1	COM1
ttyS2	COM2 (RS-485)
ttyS3	COM3
ttyS4	COM4
ttyS5	COM5
ttyS6	COM6

Table 6-4 shows the function interfaces that it provides and their roles.

Table 6-4 Uart operation functions

Functions	Description
open	open device
close	close device
read	Read characters from UART device
write	Transmit characters from UART device

6.2.1 Open and Close Function

This function interface is “Open and close device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.
For details of instructions please refer to similar functions as described in Sections 6.1.1 and 6.1.2.

The function node transferred when opening function is (/dev/ttyS0 | /dev/ttyS1 | /dev/ttyS2 | /dev/ttyS3 | /dev/ttyS4 | /dev/ttyS5 | /dev/ttyS6). The opening method is O_RDWR (read/write mode).

For details of Close function please refer to Section 6.1.2.

6.2.2 Read Function

This function interface is “read device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

1、 Prototype

```
ssize_t read(int fd, char * buf ,size_t count);
```

2. Return

If the step is successful, the system will return to the number of bytes read, If the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- char * buf

Here what is transferred is defined initial address of buffer, the data that application program reads from eeprom is saved in buf buffer.

- size_t count

Set the number of bytes that application program reads from device.

6.2.3 Write Function

This function interface is “write device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

1. Prototype

```
ssize_t write (int fd,const char * buf,size_t count);
```

2. Return

If the step is successful, the system will return to number of bytes written; if the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- char * buf

Here what is transferred is Write initial address of buffer, application program write the data of this buffer in device.

- size_t count

Set the number of bytes that application program writes in device.

6.3 Watchdog Application Development

Watchdog (WDT) is a hardware circuit which can reset system in case of system errors. In Linux system, there is one user process listening watchdog. Once system error occurs (e.g. memory error, kernel error, crash), and watchdog is not reconditioned before time is out, watchdog will reset the system (restart the system in general). User process interact with kernel drive through device file/dev/watchdog.

This function interface is mainly used to control opening and closing of watchdog, and reset watchdog counter. Table 6-5 shows the function interfaces that it provides and their roles.

Table 6-5 Description of watchdog function interface

Function	Description
open	open device
close	close device
write	Operate watchdog

6.3.1 Open and Close Function

This function interface is “Open and close device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

For details of instructions please refer to similar functions as described in Sections 6.1.1 and 6.1.2.

The function node transferred when opening function is “/dev/watchdog”. The opening method is O_RDWR (read/write mode).

For details of Close function please refer to Section 6.1.2.

6.3.2 Write Function

This function interface is “write device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

2. Prototype

```
ssize_t write (int fd,const char * buf,size_t count);
```

2. Return

If the step is successful, the system will return to number of bytes written; if the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- char * buf

Here what is transferred is Write initial address of buffer, application program write the data of this buffer in device.

- size_t count

Set the number of bytes that application program writes in device.

6.4 GPIO Application Development

This group of functions mainly provide input and output of 42 GPIOs, device node is “/dev/gpio”. Table 6-6 shows the function interfaces that it provides and their roles.

Table 6-6 Description of GPIO function interface

Function	Description
open	open device
close	close device
ioctl	GPIO opera

6.4.1 Open and Close Function

This function interface is “Open and close device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

For details of instructions please refer to similar functions as described in Sections 6.1.1 and 6.1.2.

The function node transferred when opening function is “/dev/gpio”. The opening method is O_RDWR (read/write mode).

For details of Close function please refer to Section 6.1.2.

6.4.2 Ioctl function

This function interface is “ioctl control device” as described in standard linux file operation function. User can dispatch this function using standard ioctl interface dispatch method.

1.Prototype

```
int ioctl(int fd,int cmd,long arg);
```

2. Return

if this step is successful, the system will return to “0”; if the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- int cmd

Here what is transferred is operating mode of GPIO. User can choose the options below:

GPIO_GET: get GPIO value

GPIO_SET: set GPIO output high

GPIO_CLEAR: set GPIO output low

GPIO_DIR_IN: set GPIO input

GPIO_DIR_OUT: set GPIO output

Parameters can view the file gpio_dev.h.

- long arg

arg:GPIO index number, range of 0-7

index	Hardware Pin
0	AT91_PIN_PA25
1	AT91_PIN_PA29
2	AT91_PIN_PA26
3	AT91_PIN_PA22
4	AT91_PIN_PA27
5	AT91_PIN_PB30
6	AT91_PIN_PA28
7	AT91_PIN_PC9

6.5 Beep Application Development

This group of functions mainly provide opera of BEEP, device node is “/dev/beep”. Table 6-7 shows the function interfaces that it provides and their roles.

Table 6-7 beep interface function

Function	Description
open	open device
close	close device
ioctl	GPIO operate

6.5.1 Open and Close Function

This function interface is “Open and close device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

For details of instructions please refer to similar functions as described in Sections 6.1.1 and 6.1.2.

The function node transferred when opening function is “/dev/beep”. The opening method is O_RDWR (read/write mode).

For details of Close function please refer to Section 6.1.2.

6.5.2 Ioctl Function

This function interface is “ioctl control device” as described in standard linux file operation function. User can dispatch this function using standard ioctl interface dispatch method.

1.Prototype

```
int ioctl(int fd,int cmd,long arg);
```

2. Return

if this step is successful, the system will return to “0”; if the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- int cmd

Here what is transferred is operating mode of BEEP. User can choose the options below:

```
#define IOC_BEEPDEV_MAGIC   'A'
#define AT91_BEEP_MASK      _IO(IOC_BEEPDEV_MAGIC, 1)
#define AT91_BEEP_START     _IO(IOC_BEEPDEV_MAGIC, 0)
#define AT91_BEEP_STOP      _IO(IOC_BEEPDEV_MAGIC, 1)
```

- long arg

arg:Here did not use the arg Parameters, are allowed to be empty

6.6 ADC Application Development

This group of functions provides read operation of four ADC devices. They control different ADC by different nodes. Table 6-8 shows the names of the nodes to be used in the program and the relations of hardware interfaces. For locations of each interface on SBC6020 board please refer to “Hardware manual of SBC6020”.

Table 6-8 ADC node and hardware interface

Node of device	interface
/dev/at91adc0	ADC channel0
/dev/at91adc1	ADC channel 1
/dev/at91adc2	ADC channel 2
/dev/at91adc3	ADC channel 3

Table 6-9 shows the function interfaces that it provides and their roles.

Table 6-9 ADC function interface

Function	Description
open	Open device
close	Close device
read	Read data from the ADC device

6.6.1 Open and Close Function

This function interface is “Open and close device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

For details of instructions please refer to similar functions as described in Sections 6.1.1 and 6.1.2.

The function node transferred when opening function is “ /dev/at91adc【0-3】”.The opening method is O_RDONLY (read/write mode).

For details of Close function please refer to Section 6.1.2.

6.6.2 Read Function

This function interface is “read device” as described in standard linux file operation function. User can dispatch this function using standard interface dispatch method.

2、 Prototype

```
ssize_t read(int fd, char * buf ,size_t count);
```

2. Return

If the step is successful, the system will return to the number of bytes read, If the step fails, the system will return to negative value.

3. Parameters

- int fd

Directly transfer the file descriptor got using “open” to open function.

- char * buf

Here what is transferred is defined initial address of buffer, the data that application program reads from eeprom is saved in buf buffer.

- size_t count

Set the number of bytes that application program reads from device.